60484110

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF BASIC VERSION 3

CDC® OPERATING SYSTEMS:
NOS 2
NOS/BE 1

60484110

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF BASIC VERSION 3

CDC® OPERATING SYSTEMS:
NOS 2
NOS/BE 1

REVISION RECORD

Revision	Description
A (03/19/82)	Initial release under NOS 2 and NOS/BE 1; PSR level 552.

REVISION LETTERS I, O, Q, AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Front Cover	-
Title Page	-
ii	A
iii/iv	A
v	A
vi	A
vii	A
viii	A
ix	A
1-1	A
1-2	A
2-1 thru 2-5	A
3-1 thru 3-25	A
4-1 thru 4-8	A
5-1 thru 5-23	A
A-1	A
A-2	A
B-1	A
C-1	A
C-2	A
D-1	A
D-2	A
Index-1	A
Index-2	A
Comment Sheet	A
Mailer	-
Back Cover	-

PREFACE

This manual provides the BASIC programmer with assistance in the debugging of BASIC Version 3 programs under the control of the CDC® CYBER Interactive Debug Facility.

CYBER Interactive Debug (CID) operates under the following operating systems:

NOS 2 for the CONTROL DATA® CYBER 170 Computer Systems; CYBER 70 Computer System models 71, 72, 73, 74; and 6000 Computer Systems

NOS/BE 1 for the CDC CYBER 170 Computer Systems; CYBER 70 Computer System models 71, 72, 73, 74; and 6000 Computer Systems

You should have a copy of the CYBER Interactive Debug reference manual available for reference, but you need not be familiar with the manual. In addition, you should be familiar with BASIC 3 and should be able to run jobs interactively under either NOS Interactive Facility or the NOS/BE INTERCOM.

This guide provides a tutorial approach to CID beginning with fundamental features and proceeding through more advanced features. Section 1 provides some background information and presents a summary of the features of CID. Section 2 describes the method for initiating a debug session with CID, and describes several useful CID commands; this section contains sufficient information to allow the less experienced user to make productive use of CID. Sections 3 through 5 describe features which are helpful in debugging more complex programs. This user's guide is not comprehensive in its approach to CID; only those features considered useful to BASIC programmers are described. Most of the features described in this user's guide are illustrated by actual examples of debug sessions. This is intended to help you become familiar with CID notational conventions and with information produced by CID.

Additional information can be found in the publications listed below.

The NOS Manual Abstracts and the NOS/BE Manual Abstracts are instant-sized manuals containing brief descriptions of the contents and intended audience of all NOS and NOS product set manuals, and NOS/BE and NOS/BE product set manuals, respectively. The abstracts manuals can be useful in determining which manuals are of greatest interest to a particular user. The Software Publications Release History serves as a guide in determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software.

The following manuals are of primary interest:

Publication	Publication Number
CYBER Interactive Debug Version 1 Reference Manual	60481400
BASIC Version 3 Reference Manual	19983900

The following manuals are of secondary interest:

Publication	Publication Number
CYBER Loader Version 1 Reference Manual	60429800
INTERCOM Version 5 Reference Manual	60455010
NOS Version 2 Manual Abstracts	60485500
NOS Version 2 Reference Set, Volume 3 System Commands	60459680
NOS/BE Version 1 Manual Abstracts	84000470
Software Publications Release History	60481000
XEDIT Version 3 Reference Manual	60455730

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This manual describes a subset of the features and parameters documented in the CYBER Interactive Debug Version 1 Reference Manual and the BASIC Version 3 Reference Manual. Control Data cannot be responsible for the proper functioning of any features or parameters not documented in the CYBER Interactive Debug Version 1 Reference Manual.

CONTENTS

FIGURES

TABLES

NOTATIONS

Certain notations are used throughout this manual. The notations and their meanings are:

...	Horizontal ellipses indicate repetition.
UPPERCASE	Uppercase text in examples of terminal dialog indicates terminal output. Uppercase words in command formats must appear exactly as shown. Only uppercase words are used in command format examples.
lowercase	Lowercase text in examples of terminal dialog indicates user input. Lowercase words in command formats indicate values or options supplied by the user.

Examples of actual terminal sessions appearing in this manual were produced on a class 1 terminal. The format of these terminal sessions might differ slightly from the formats appearing at your terminal.

INTRODUCTION 1

The CYBER Interactive Debug facility (CID) allows you to interactively debug your executing BASIC program. CID can be used with BASIC 3 programs compiled under the NOS or NOS/BE operating systems.

Use of CID requires a mode of execution called debug mode which is activated by a system control statement. As long as debug mode is in effect, execution of all your programs takes place under CID control. CID allows you to enter commands that perform the following operations:

Suspend program execution at specified locations

Suspend program execution on the occurrence of selected conditions, such as modification of a variable

Display the values of simple or subscripted variables while execution is suspended

Change the values of simple or subscripted variables while execution is suspended

Resume program execution at the location where it was suspended or at another location

WHAT IS INTERACTIVE DEBUGGING?

Interactive debugging means that you debug your program while it is executing. In interactive mode, CID allows you to suspend execution of your program and enter commands directly from a terminal while execution is suspended. CID executes each command immediately after it is entered. Program execution remains suspended until resumed by the appropriate command. In this manner, you can control and monitor the execution of your program, stopping at desired points to examine and modify the values of program variables.

WHY USE CID?

Debugging often requires recompiling a program several times to make corrections or to add statements that print intermediate values of program variables. This debugging technique can be expensive in terms of both machine and programmer time.

CID, however, allows you to debug a program by referring only to the source listing and by referencing variables and line numbers symbolically. Since CID allows you to make changes to your program's data and flow of control as execution proceeds, you can often accomplish in a single session debugging that would normally require several compilations. Thus, considerable time can be saved, especially when you are debugging programs that are time-consuming to compile and execute.

SPECIAL CID FEATURES FOR BASIC PROGRAMS

When the BASIC source program is compiled for use with CID, additional tables are output with the object code. These additional tables provide the name, relative address, type, and dimension of every program variable, and the relative locations of all executable statements.

The table information is used by CID to provide both the operations listed previously and certain features currently available only to BASIC programs compiled for use with CID. These features include commands with a BASIC-like syntax and the capability of symbolically referencing locations within an object program. The commands available only to programs compiled for use with CID are indicated in appendix D.

The BASIC-like syntax commands are PRINT, LET, GOTO, MAT PRINT, and IF. These commands have the same syntax and function as equivalent BASIC statements except for the following restrictions (and those noted in the command descriptions):

Arithmetic or string expressions cannot refer to system or user-defined functions.

Arithmetic expressions cannot contain the exponentiation operator ^.

Multiple CID commands can appear on the same line if they are separated by one or more semicolons. However, because items in a BASIC print list can be separated by a semicolon, two semicolons must separate a PRINT or MAT PRINT command from any following commands.

The main advantage of the BASIC-like CID commands, apart from their familiarity to BASIC programmers, is that they provide automatic output formatting or referencing by variable name.

Expressions used with these BASIC-like commands follow the syntax and operator precedence rules of BASIC expressions except that function references and exponentiation are not allowed.

For purposes of this user's guide, it is assumed that BASIC programs to be executed under CID control are compiled for use with CID, which makes the special features available to BASIC programs. Therefore, in the discussions of the CID capabilities, no distinction is made between standard CID features and the special features available to BASIC programs. It is possible, though more difficult, to use CID with programs not compiled in debug mode. See the CYBER Interactive Debug reference manual and the BASIC reference manual for a description of this capability.

EFFECTS OF CID ON PROGRAM SIZE AND EXECUTION TIME

CID affects your program's field length in the following manner. CID consists of several parts that are similar to overlays. The main part is always in memory and is approximately 4000_8 words long. The other parts are exchanged in memory with the program being debugged and can require up to 54000_8 words of memory. Therefore, if your program is smaller than 54000_8 words, the field length requirement when your program is debugged under CID is approximately 60000_8 words. If your program is larger than 54000_8 words, the field length requirement is 4000_8 words larger than the size of your program.

Certain CID features require a mode of execution called interpret mode (described in section 3), which requires much more execution time than normal execution. This can be a significant problem in some programs. In some cases, however, you can substitute an alternate feature that does not require interpret mode.

PROGRAMMING FOR EASE OF DEBUGGING

Even though CID offers many useful features, a well-designed program is much easier to debug than a poorly-designed program. When designing your program, make sure you understand what the program is supposed to do before deciding how the program will do it.

Using a style of coding that avoids GOTOs and minimizes branches can help in the debugging process. A program that contains a minimum of branches and flows logically from top to bottom is much easier to understand than one that contains many unnecessary branches. CID provides features that allow you to trace the flow of control of your executing program; this process is much easier if the program avoids needlessly complex logic.

You should avoid programming tricks and shortcuts, particularly if they depend on system idiosyncracies.

CID should not be considered a substitute for proper programming practices. Program carefully and try to minimize the number of errors. Performing a careful visual scan of the program before execution can reveal many of the more obvious errors and can reduce the amount of time spent debugging your program.

BATCH MODE DEBUGGING

Although CID is intended to be used interactively, it can be used in batch mode. Batch mode debugging is described in appendix C.

This section summarizes the operations necessary to conduct a debug session and introduces several CYBER Interactive Debug (CID) notation conventions. At the end of the section, several fundamental commands are presented and used in a sample debug session. These commands enable you to conduct a simple, but useful, debug session.

BEGINNING A DEBUG SESSION

To execute a program under CID control (and to make use of the BASIC capabilities), you must compile and execute the program in debug mode. You turn on debug mode with a system control statement.

DEBUG CONTROL STATEMENT

The DEBUG control statement activates debug mode. The format of this statement is as follows:

```
DEBUG
```

or

```
DEBUG(ON)
```

When a BASIC program is compiled in debug mode, special symbol tables for use by CID are generated as part of the object code. When the program is subsequently executed in debug mode, all of the CID features can be used. Note that a program which has not been compiled with debug mode activated can still be executed in debug mode, but some of the features described in this user's guide will not be available.

When debug mode is on, you can interact with the operating system and perform all other terminal activities in a normal manner.

The statement to deactivate debug mode is as follows:

```
DEBUG(OFF)
```

After debug mode is turned off, programs that were compiled in debug mode will be executed in normal, non-debug mode. Debug mode should be turned off only if you do not wish subsequent compilations and executions to occur under CID control.

EXECUTING UNDER CID CONTROL

A debug session consists of the sequence of interactions between you and CID that occurs while your object program is executing in debug mode. The session begins when you initiate execution of your object program and ends when you enter the QUIT command. For an explanation of the QUIT command, see Some Essential Commands in this section.

To initiate a debug session, compile and execute your program in a normal manner. The system loads the CID program module, your binary program, and system and library modules. Control then transfers to an entry point in CID and CID issues the message:

```
CYBER INTERACTIVE DEBUG
?
```

The ? character is a prompt signifying that CID is waiting for user input. At this point you can enter CID commands.

The examples in figure 2-1 show the statements necessary for compiling a program and initiating a debug session under the NOS (examples 1 and 2) and NOS/BE (examples 3 and 4) operating systems. In this figure and in all terminal sessions in this guide, user input is in lowercase and system response is in uppercase.

Debugging a program can require more than one debug session. If this is the case, you can terminate the current debug session and initiate a new session.

ENTERING CID COMMANDS

The CID prompt for user response is a question mark. In response to the ? character, enter a CID command and press the transmission key (RETURN on most terminals). CID then processes the command, issues an informative message indicating the disposition of the command, or displays any output that the command calls for and issues another ? prompt. CID continues to issue prompts after processing commands until you enter the command to resume execution of your program or until you terminate the session.

If you enter a command incorrectly, CID displays a diagnostic message. One such message is as follows:

```
*ERROR - UNKNOWN COMMAND
```

If this message appears, determine the correct format and reenter the command. You can use the HELP command, described later in this section, for assistance with command formats. For a complete listing of CID diagnostics, see the CYBER Interactive Debug reference manual.

```
EXAMPLE 1:

     /basic ◄──────────────────────────── Enter BASIC subsystem.
     OLD, NEW, OR LIB FILE: old,proga ──── Designate PROGA as primary file.

      READY.
     debug ◄──────────────────────────── Activate debug mode.

      READY.
     run ◄──────────────────────────── Compile program and initiate debug session.

      CYBER INTERACTIVE DEBUG
     ?


EXAMPLE 2:

     /debug(on) ◄──────────────────────────── Activate debug mode.
     /x,basic,i=proga,b=basfil ◄──────────── Compile program.
     /basfil ◄──────────────────────────── Execute program and initiate debug session.

     CYBER INTERACTIVE DEBUG
     ?


EXAMPLE 3:

     COMMAND- editor ◄──────────────────── Enter edit mode.
     ..format,basic ◄──────────────────── Request BASIC format specifications.
     ..edit,proga ◄──────────────────── Make PROGA the edit file.
     ..debug ◄──────────────────── Activate debug mode.
     ..run,basic ◄──────────────────── Compile program and initiate debug session.

     CYBER INTERACTIVE DEBUG
     ?


EXAMPLE 4:

     COMMAND- debug ◄──────────────────── Activate debug mode.
     COMMAND- basic,i=proga ◄──────────── Compile program and initiate debug session.

     CYBER INTERACTIVE DEBUG
     ?
```

Figure 2-1. Initiating a Debug Session

SHORTHAND NOTATION FOR CID COMMANDS

Most standard CID commands have a shorthand form that allows you to omit the comma separator and to substitute abbreviations for the command name and certain parameters. For example, the command

```
SET,BREAKPOINT,L.250
```

can be abbreviated as follows:

```
SB L.250
```

In this guide, both short and long command forms are described. However, to make sample debug sessions as understandable as possible, long forms are shown. You are encouraged to use the short forms as you become familiar with CID; they have the same effect as long forms.

A more detailed explanation of CID command syntax and a list of long and short command forms are given in appendix D.

MULTIPLE COMMAND LINES

You can enter several CID commands on the line if you separate them with semicolons. For example, entering

```
SET,BREAKPOINT,L.210;GO
```

has the same effect as entering

```
SET,BREAKPOINT,L.210
GO
```

Note that two semicolons must separate the PRINT (described later in this section) or the MAT PRINT (see section 3) commands from the following command.

REFERENCING SOURCE STATEMENTS BY LINE NUMBER SPECIFICATION

Many of the CID command formats require that you indicate a specific statement within the program you are debugging. Source statements are referenced by line number using the notation

```
L.n
```

where n is the statement line number. This notation denotes the source line having the specified line number. Leading zeros can be omitted. Some examples of line number references are as follows:

```
L.130

L.510

L.260
```

SOME ESSENTIAL COMMANDS

The following paragraphs describe several CID commands you can use to conduct simple debug sessions. These commands are the GO command, QUIT command, PRINT command, and SET,BREAKPOINT command. (These commands are described in greater detail in section 3.) The HELP command, which provides a quick summary of information about various CID subjects, is also described.

The command forms presented here allow you to debug only single unit BASIC programs (BASIC programs which do not contain any FORTRAN subroutine calls). To debug BASIC programs containing multiple program units (BASIC main programs which call FORTRAN subroutines), you must be familiar with the home program concept described in section 4.

GO

The command to initiate or resume program execution is as follows:

```
GO
```

If entered at the beginning of the debug session, this command initiates program execution. If entered after execution has been suspended, this command causes execution to resume at the statement where it was suspended.

Once execution of your program has been suspended, you can enter any number of CID commands. Execution remains suspended until you enter GO.

QUIT

The command to terminate a debug session is as follows:

```
QUIT
```

In response to the QUIT command, the following message is displayed under the NOS BATCH subsystem and NOS/BE:

```
DEBUG TERMINATED
```

The following message is displayed under the NOS BASIC subsystem:

```
SRU       n.nnn UNTS.

RUN COMPLETE
```

The QUIT command causes an exit from the current session and a return to system command mode. Files accessed by the BASIC program are closed. Note, however, that debug mode remains on until DEBUG(OFF) is specified.

Breakpoints and other alterations to the object program exist only for the duration of the debug session. You can terminate a debug session any time you have control (CID has issued a ? prompt). The object program can then be executed normally, or it can be executed again under CID control.

PRINT

CID provides several commands for displaying the values of program variables. The simplest and most useful of these is the command

```
PRINT output-list
```

where output-list is a list of any number of restricted arithmetic or string expressions that are separated by commas or semicolons. Restricted expressions are BASIC expressions that do not refer to system or user-defined functions, or do not contain the exponentiation operator.

This command lists the values of the specified program variables. Values are formatted according to type (numeric or string).

Some examples of the PRINT command are as follows:

```
PRINT A

PRINT B$(3:4)

PRINT S(10)
```

SET,BREAKPOINT

A breakpoint is a location within a program where execution is to be suspended. The command to establish a breakpoint has the form

```
SET,BREAKPOINT,L.n
```

where L.n is a line number specification as described earlier in this section under Referencing Source Statements by Line Number Specification. The short form of SET,BREAKPOINT is SB.

Examples of the SET,BREAKPOINT command are as follows:

```
SET,BREAKPOINT,L.140
```

Sets a breakpoint at line 140.

```
SB,L.210
```

Sets a breakpoint at line 210.

You can establish breakpoints at any time in the debug session when execution is suspended and CID has issued a ? prompt. A breakpoint can be established at any executable statement. Only one breakpoint can be set at a single statement.

When a breakpoint is encountered, CID receives control and issues the message

```
*B #i AT L.n
```

where i is a breakpoint number assigned by CID, and n is the number of the line where the breakpoint was set. Breakpoints are assigned numbers by CID when they are established. Up to sixteen breakpoints can be in effect at the same time.

Establishing a breakpoint at a specified location does not alter execution of the statement at that location. When a breakpoint is encountered during execution, control transfers to CID, which then allows you to enter CID commands. Typically, commands are entered to examine the values of program variables, and then execution is resumed. When execution is resumed, execution begins with the statement at the breakpoint location.

HELP

CID provides a HELP command that displays a brief summary of information about specific CID subjects and commands. You can issue the HELP command whenever you need assistance with a particular aspect of CID.

Simply entering the command

```
HELP
```

causes CID to display a list of subjects. To obtain additional information about any subject in the list, enter the following command:

```
HELP,subject
```

For example, the command HELP,ERROR displays a brief description of error processing.

A useful form of the HELP command is HELP,CMDS. This command displays a complete list of CID commands and a brief explanation of each. You can obtain a more detailed explanation of any CID command by entering

```
HELP,command
```

where command is any CID command. The HELP command does not provide the same level of detail as the CID reference manual, however, and should not be considered a substitute for the reference manual.

The HELP command is illustrated in figure 2-2, which shows the entry of the command HELP,SET,BREAKPOINT to display a summary of the command parameters.

SUMMARY

To use CID, as presented in this section, follow this step-by-step procedure:

1. Type DEBUG to turn on debug mode.
2. Compile and execute your program in a normal manner. Control transfers to CID when execution begins. CID displays a message at the terminal and waits for your input.
3. Set breakpoints as desired.

   To set a breakpoint at a line number, enter

   ```
   SET,BREAKPOINT,L.n
   ```

   where n is a line number.
4. Enter GO to begin program execution.

   CID executes your program in a normal manner, but returns control to you when a breakpoint occurs.

```
CYBER INTERACTIVE DEBUG
? help,set,breakpoint
SB  -  SET  BREAKPOINT - ALLOWS YOU TO SET A BREAKPOINT AT A
SPECIFIC LOCATIONS IN USER'S PROGRAM.  THE FORM OF  THE  SET
BREAKPOINT COMMAND IS.
     SB <LOCATION>,<FIRST>,<LAST>,<STEP>
WHERE <LOCATION> IS THE LOCATION IN YOUR PROGRAM AT WHICH
YOU WANT THE BREAKPOINT SET.
<FIRST>, <LAST> AND <STEP> ARE OPTIONAL AND ARE DEFAULTED TO
1, 131071 AND 1 RESPECTIVELY.  THE BREAKPOINT IS NOT HONORED
UNTIL <LOCATION> HAS BEEN HIT <FIRST> TIMES. BUT, IT WILL BE
HONORED WHEN <LOCATION> IS HIT THE <FIRST>TH TIME AND EACH
<STEP>TH TIME AFTER THAT AS LONG AS <LAST> IS NOT EXCEEDED.
IF YOU TERMINATE THE SB COMMAND WITH AN OPEN BRACKET [, THEN
ALL COMMANDS UP TO A CLOSE BRACKET ] WILL BE COLLECTED  SUCH
THAT  WHEN THE BREAKPOINT IS HONORED, THOSE COMMANDS WILL BE
EXECUTED.
?
```

Figure 2-2. Example of HELP Command

5. When control returns to you, the values of the program variables can be displayed by entering the command:

   PRINT output-list

   Enter GO to resume program execution.

6. Enter QUIT to terminate the session. Enter DEBUG(OFF) to turn off debug mode.

Debug sessions can become complicated; however, you should always try to keep debug sessions short and simple. If necessary, correct known bugs, recompile your program, and conduct additional debug sessions.

SAMPLE DEBUG SESSION

The preceding commands are now used to conduct a sample interactive debug session. As you study each example in this guide, keep in mind that the purpose of the examples is to illustrate the previously mentioned CID features; the examples are not intended to present a suggested sequence of commands for debugging all programs. The actual commands used in a given debug session depend on your program and, often, on your intuition.

A BASIC program and a debug session log are illustrated in figure 2-3. The program reads the values for numeric variables A and B, calculates the value of A times the square root of B, then assigns this value to numeric variable C. Finally, the value of C is printed. After the debug session is initiated, a breakpoint is set at line number 120. When line 120 is reached during execution, CID gains control and prompts for user input. The PRINT command is entered to display the values assigned to the variables. Note that the value of variable C is zero because the breakpoint suspended program execution before the assignment statement for variable C (line 120) was executed. Execution is then resumed with the GO command and the program runs to completion. (The END trap, which occurs on normal program termination, is described in section 3.) Another PRINT command is entered to display the values of the variables. At this point, the assignment statement at line 120 has been executed, so the value of variable C is no longer zero.

```
Program VALUE:

     00100 REM PROGRAM VALUE
     00110 READ A,B
     00120 LET C=A*SQR(B)
     00130 PRINT C
     00140 DATA 300,500
     00150 END


Session Log:

      CYBER INTERACTIVE DEBUG
     ? set,breakpoint,l.120 <------ Set breakpoint at line 120.

     ? go <------------------------ Initiate execution.

      *B #1, AT L.120 <------------ Breakpoint detected at line 120.
     ? print a;b;c <-------------- Display values of variables A, B, and C.

       300  500  0
     ? go <------------------------ Resume execution.

      6708.2 <--------------------- Program output.
      *T #17, END IN L.150 <------- Program runs to completion.
     ? print a;b;c <-------------- Oisplay final values of variables A, B, and C.

       300  500  6708.2
     ? quit <--------------------- Terminate debug session.
```

Figure 2-3. Sample Debug Session

Once you have compiled your BASIC program in debug mode and have initiated a debug session, you are ready to begin interactive debugging. Program execution under CYBER Interactive Debug (CID) control involves interaction between you and CID. You specify conditions for which program execution is to be suspended, and CID gives you control when these conditions are satisfied and allows you to enter various CID commands to examine and alter the status of your program.

The preceding section presented some elementary commands you can use to conduct a simple debug session. This section presents additional information about the commands introduced in section 2 and describes other commands and CID features that allow you to use CID more productively. The commands discussed in this section enable you to do the following:

Suspend program execution; the commands are SET,BREAKPOINT and SET,TRAP.

Display the current values of program variables and arrays at the terminal while execution is suspended; the commands are PRINT, MAT PRINT, LIST,VALUES, and DISPLAY.

Alter the contents of simple and subscripted variables; the command is LET.

Alter the flow of program execution; the command is GOTO.

ERROR AND WARNING PROCESSING

Each time you enter a command, CID checks the command for errors. If errors are detected, CID issues an error. If a questionable situation exists, CID issues a warning message.

ERROR MESSAGES

CID issues an error message whenever it encounters a command that cannot be executed. Error messages are usually caused by a misspelled command or an illegal or misspelled parameter. CID does not attempt to execute an erroneous command; instead, CID issues an error message followed by a ? prompt. The format of the error message is as follows:

```
*ERROR - message
?
```

The message contains a brief description of the error. In response to an error message, you should consult the CID reference manual or use the HELP command to determine the correct command form, and reenter the command.

Figure 3-1 illustrates two error messages. The first message is caused by a misspelled PRINT command. The second message is caused by the GO command being entered after program execution has terminated.

```
 *T #17, END IN L.170
? prnit a
 *ERROR - UNKNOWN COMMAND
? print a
  2
? go
 *ERROR - PROGRAM HAS COMPLETED
?
```

Figure 3-1. Debug Session Illustrating Error Messages

WARNING MESSAGES

CID issues a warning message if the command you have entered will have consequences you might not be aware of or if the command will result in CID action other than that which you have specified. The warning message is followed by a special input prompt; in response to this prompt, you can tell CID either to execute the command or to ignore it. The format of a warning message is:

```
*WARN - message
OK?
```

The message describes the action CID will take if allowed to execute the command. In response to a warning message you can enter the following:

YES or OK

CID executes the command.

NO

CID disregards the command.

Any CID Command

CID disregards the previous command and executes the new one.

Warning messages can be suppressed by an option on the SET,OUTPUT command (described in section 4 under Control of CID Output). In this case, CID automatically takes the action indicated in the message without providing notification.

Figure 3-2 illustrates two warning messages. The first message occurs when a SET,BREAKPOINT command is entered for a line beyond the last executable statement of the program. In answer to the prompt, OK is entered which causes a breakpoint to be set at line 250. The second message occurs when a form of the CLEAR,BREAKPOINT command is entered. In answer to the prompt, NO is entered which means that no breakpoints will be cleared.

See the CYBER Interactive Debug reference manual for a complete list of warning messages and an explanation of each.

```
? set,breakpoint,l.260
 *WARN - LINE 260 NOT EXECUTABLE - LINE 250 WILL BE USED
 OK ? ok
? clear,breakpoint
 *WARN - ALL WILL BE CLEARED
 OK ? no
?
```

Figure 3-2. Debug Session Illustrating Warning Messages

BREAKPOINTS AND TRAPS

When conducting a debug session, you must initially provide for gaining interactive control at some point within your program. CID provides two methods of doing this: breakpoints and traps.

A breakpoint (introduced in section 2) causes suspension of program execution when a specified line number is reached during execution. A trap causes suspension of program execution when a specified condition is detected during execution. Both breakpoints and traps cause CID to give control to you so that you can examine and alter the status of your program at various points during execution.

In a typical debug session, you establish breakpoints and traps before initiating program execution. When a breakpoint is detected or a trap condition occurs during execution, CID receives control and, in turn, allows you to enter CID commands.

In most debugging situations, breakpoints, rather than traps, are recommended for suspending execution. Breakpoints allow you to suspend execution at any executable statement in your program and can, in most cases, be substituted for traps. Traps can be useful in certain cases, but some trap types require that you understand compiler-generated object code; only trap types useful to most BASIC programmers are described here.

Breakpoints and traps exist until explicitly removed with a clear command or until the debug session is terminated. An object program is not permanently altered by any breakpoints or traps established during a session.

CID provides commands that enable you to:

Set breakpoints and traps

List existing breakpoints and traps

Clear existing breakpoints and traps

Save breakpoint and trap definitions on a separate file for use in a later debug session

SUSPENDING EXECUTION WITH BREAKPOINTS

A breakpoint is established at a specified location within a program such that when the location is reached during program execution, control passes to CID. CID displays a message and gives control to you.

You can use the SET,BREAKPOINT command (introduced in section 2) to set breakpoints at any executable statement in your program. For example,

```
SET,BREAKPOINT,L.100
```

sets a breakpoint at line 100 of your program.

It is important to note that breakpoints suspend execution before the statement is executed. For example, assume a program contains the statements

```
200     LET A=0.0
210     LET A=A+1.0
```

and that a breakpoint is set at line 210. When line 210 is reached, execution is suspended immediately; therefore, line 210 is not executed. Thus, A has the value zero, rather than one. When execution is resumed, the statement at line 210 is executed and the value of zero is replaced by one.

FREQUENCY PARAMETERS

When a breakpoint is set at a line number, execution is suspended each time that line is reached. For example, if a breakpoint is set at a line number within a loop, suspension occurs on each pass through the loop. This can result in many unnecessary breaks during the course of a debug session. To alleviate this situation, CID provides another form of the SET,BREAKPOINT command that is useful for debugging loops and other sections of a program that are executed frequently. The form of this command is

```
SET,BREAKPOINT,L.n,first,last,step
```

where first, last, and step are frequency parameters. The parameter first indicates the first time the breakpoint suspends execution. The parameter last indicates the last time the breakpoint suspends execution. The parameter step indicates how often the breakpoint suspends execution. For example, the command

```
SET,BREAKPOINT,L.150,10,100,5
```

sets a breakpoint at line 150. The breakpoint is effective on the tenth time the statement is reached and every fifth time thereafter, up through the hundredth time.

As an example of the use of the frequency parameters, consider the following loop:

```
150 FOR I=1 TO 1000
160 LET X=X*I
170 NEXT I
```

To examine the progress of the iteration X=X*I, you can set a breakpoint at line 170, specifying frequency parameters to suspend execution at an interval rather than on each pass through the loop. For example,

```
SET,BREAKPOINT,L.170,3,1000,100
```

sets a breakpoint that suspends execution on every hundredth pass through the loop starting with the third pass and ending with the nine hundred and third pass.

To illustrate the SET,BREAKPOINT command, the program shown in figure 3-3 is executed under CID control. The program TRIANGL, which calculates the area of a triangle, reads input data from the file TRARDAT. Subroutine AREA then performs the computation and returns the final result. Control branches to the beginning of the program, and another record is read and processed. A sample input file with four lines is also shown in figure 3-3.

```
Program TRIANGL:

    00100 REM PROGRAM TRIANGL
    00110 FILE #1="TRARDAT"
    00120 IF END #1 GOTO 00170
    00130 INPUT #1,X1,Y1,X2,Y2,X3,Y3
    00140 GOSUB 00190
    00150 PRINT "THE AREA OF THE TRIANGLE IS ";A
    00160 GOTO 00120
    00170 STOP
    00180 REM SUBROUTINE AREA
    00190 LET S1=SQR((X2-X1)^2+(Y2-Y1)^2)
    00200 LET S2=SQR((X3-X1)^2+(Y3-Y1)^2)
    00210 LET S3=SQR((X3-X2)^2+(Y3-Y2)^2)
    00220 LET T=(S1+S2+S3)/2.0
    00230 LET A=SQR(T*(T-S1)*(T-S2)*(T-S3))
    00240 RETURN
    00250 END


Input Data:

    0.0 0.0 2.0 0.0 0.0 2.0
    0.0 1.0 0.5 2.0 -1.0 1.2
    0.2 -2.9 -1.3 8.0 5.6 7.8
    6.1 2.0 0.1 -4.0 3.2 7.0
```

Figure 3-3. Program TRIANGL and Input Data

A debug session using the program and data in figure 3-3 is shown in figure 3-4. In this session, execution is suspended in the program immediately before the control branches to subroutine AREA in order to examine the input values. Execution is also suspended at the end of subroutine AREA to examine the intermediate values and the final result. To accomplish this, breakpoints are set at line 140 and line 240 of the program. In both SET,BREAKPOINT commands, the frequency parameters 1,10,2 are included so that execution is suspended only on every other pass through the program, beginning with the first pass. After the tenth pass, the breakpoint is not recognized. Because only four input lines are contained in file TRARDAT, execution is suspended on the first and third passes through the program. Each time execution is suspended, the PRINT command is entered to display the desired values, and the GO command is entered to resume execution.

LISTING BREAKPOINTS

You can display a list of breakpoints defined in a debug session by entering the command:

```
LIST,BREAKPOINT,*
```

This command displays a list of all breakpoints in the program. The short form of LIST,BREAKPOINT is LB.

The LIST,BREAKPOINT command lists the breakpoints that exist when the command is entered. The list contains the number and location of each breakpoint in the form

```
*B #i = L.n
```

where i is the breakpoint number assigned by CID. If frequency parameters were specified when the breakpoint was set, they also appear in the list.

You can display a specific breakpoint by entering the command

```
LIST,BREAKPOINT,loc
```

where loc is either a line number (L.n) or a breakpoint number (#n).

If no breakpoints exist when a LIST,BREAKPOINT command is entered, CID displays the following message:

```
NO BREAKPOINTS
```

Examples of the LIST,BREAKPOINT command are as follows:

```
LIST,BREAKPOINT,#5
```

Lists breakpoint number 5.

```
LB,L.150
```

Lists breakpoint at line 150.

Figure 3-5 illustrates a debug session for the program shown in figure 3-3 in which some breakpoints are defined and are listed later in the session.

CLEARING BREAKPOINTS

When breakpoints are no longer necessary, you can clear all breakpoints or individual breakpoints by entering one of following forms of the CLEAR,BREAKPOINT command. The form

```
CLEAR,BREAKPOINT,*
```

clears all breakpoints in the program.

If you do not include the asterisk and enter only

```
CLEAR,BREAKPOINT
```

CID displays the message:

```
*WARN - ALL WILL BE CLEARED
OK?
```

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.140,1,10,2 <-- Set breakpoint at line 140.  Breakpoint suspends
                                  execution on first and third passes.

? set,breakpoint,l.240,1,10,2 <-- Set breakpoint at line 240.  Breakpoint suspends
                                  execution on first and third passes.

? go <-- Initiate execution.

 *B #1, AT L.140 <-- Execution suspended at line 140.
? print x1;y1;x2;y2;x3;y3 <-- Display first set of input values.

  0  0  2  0  0  2
? go <-- Resume execution.

 *B #2, AT L.240 <-- Execution suspended at line 240.
? print s1;s2;s3;t;a <-- Display intermediate values and final result.

  2  2  2.82843  3.41421  2
? go <-- Resume execution.

THE AREA OF THE TRIANGLE IS  2 <-- Program output.
THE AREA OF THE TRIANGLE IS  .55 <-- Second record of input data executed.
 *B #1, AT L.140 <-- Execution suspended at line 140.
? print x1;y1;x2;y2;x3;y3 <-- Display third set of input data.

  .2 -2.9 -1.3  8  5.6  7.8
? go <-- Resume execution.

 *B #2, AT L.240 <-- Execution suspended at line 240.
? print s1;s2;s3;t;a <-- Display intermediate values and final result.

  11.0027  11.9854  6.9029  14.9455  37.455
? go <-- Resume execution.

THE AREA OF THE TRIANGLE IS  37.455 <-- Program output.
THE AREA OF THE TRIANGLE IS  23.7 <-- Fourth record of input data executed.
 *T #17, END IN L.170 <-- Program runs to completion.
? quit <-- Terminate session.
```

Figure 3-4. Debug Session Illustrating SET,BREAKPOINT Command

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.140,1,10,2 <-- Set breakpoint at line 140.

? set,breakpoint,l.240,1,10,2 <-- Set breakpoint at line 240.

? go <-- Initiate execution.

 *B #1, AT L.140 <-- Breakpoint detected at line 140.
? list,breakpoint,* <-- List all breakpoints.

 *B #1 = L.140,,10,2,   *B #2 = L.240,,10,2
?
```

Figure 3-5. Debug Session Illustrating LIST,BREAKPOINT Command

This message serves as a reminder that the command you have just entered will remove all breakpoints in the entire program. If this is not what you want, enter

```
NO
```

and CID disregards the CLEAR,BREAKPOINT command. If you do want the CLEAR,BREAKPOINT command to be executed, enter

```
OK
```

and CID clears all existing breakpoints.

The form

```
CLEAR,BREAKPOINT,loc-list
```

clears the specified breakpoints; loc-list is a list of locations which are separated by commas. Each location has one of the following forms:

L.n	Line n of the program
#n	Breakpoint having number n

If a breakpoint does not exist at a specified location, CID displays the message

```
NO BREAKPOINT loc
```

where loc is the breakpoint location.

The short form of CLEAR,BREAKPOINT is CB.

Some examples of the CLEAR,BREAKPOINT command are as follows:

```
CLEAR,BREAKPOINT,L.140,L.200
```

Removes breakpoints from lines 140 and 200.

```
CLEAR,BREAKPOINT,#3,#5,#6
```

Removes breakpoints 3, 5, and 6.

```
CB,L.350,L.460
```

Removes breakpoints from lines 350 and 460.

SUSPENDING EXECUTION WITH TRAPS

Traps suspend execution and give you control whenever specified conditions occur. For example, traps can give you control when you enter a terminal interrupt, when execution is terminated, or when the beginning of a new line is reached.

TRAP USAGE

The traps most useful to the BASIC programmer are the LINE and STORE traps. (The END, ABORT, and INTERRUPT traps are also used, but they are established automatically by CID.) The remaining CID traps should not be used with BASIC programs because their use can be time-consuming, they are aimed at machine language programmers, and they can cause BASIC program execution to be suspended in unexpected places. The traps described in this section are listed in table 3-1. See the CYBER Interactive Debug reference manual for information about other CID traps.

When a trap condition is detected, execution is suspended. CID gains control and issues a message identifying the trap, followed by a ? prompt for user input. The general format of the trap message is as follows:

```
*T #i, type AT L.n
```

where i is the trap number which is assigned by CID when the trap is set, type briefly describes the condition that caused the trap, and n is the line in the program where execution was suspended. If IN rather that AT is specified, then execution was suspended during and not before execution of the indicated line.

An example of a trap message is as follows:

```
*T #3, LINE AT L.150
?
```

In this example, a LINE trap has been detected at line 150 of the program; this trap was the third one established.

In response to the ? prompt, you can enter any CID command. Typically, you will use this opportunity to examine the values of program variables and make any desired changes to these values. You can resume program execution by entering the GO command.

Traps suspend execution when a specific event occurs. Some traps suspend execution before the event, while others suspend execution after the event. This is an important distinction because it can affect the status of variables you are displaying or altering. For example, assume that execution is suspended at line 210 of the following program segment:

```
200     LET A = 0.0
210     LET A = A+1.0
```

If the trap suspends execution before the statement at line 210 is executed, the value of A is zero. If the trap suspends execution after the statement is executed, the value of A is one.

Table 3-1 indicates, for each trap, the point in execution where CID gets control.

The traps described in this section are of two types: default and user-established. The default traps always exist; you need not specify a SET,TRAP command for these traps. You set the user-established traps with the SET,TRAP command. Table 3-1 indicates default and user-established traps.

TABLE 3-1. TRAP TYPES

Trap Type	Short Form	Condition	Established By	User Gets Control
LINE	L	Beginning of an executable statement	User	Before the statement is executed
STORE	S	Store to memory	User	After the store
INTERRUPT	INT	User interrupt	Default	After the interrupt
END	E	Normal program termination	Default	After termination
ABORT	A	Abnormal program termination	Default	After termination

DEFAULT TRAPS

CID provides default traps that are automatically set at the beginning of a debug session. These traps allow you to gain control without explicitly establishing any breakpoints and traps. The default traps are the END, ABORT, and INTERRUPT traps.

Together, the END and ABORT traps transfer control to CID when program execution terminates. Thus, for the initial debug session, you can allow your program to terminate; then by examining the status of the program at the point of termination, you can determine where breakpoints or traps should be set for subsequent sessions.

END Trap

The END trap gives control to CID when program execution terminates normally, regardless of any CID commands that have been entered to set or clear traps.

The general format of the END trap message is as follows:

```
*T #17, END IN L.n
?
```

where n is the line in which program execution terminated. CID permanently assigns the number 17 to the END trap.

Note that the debug session does not end when execution of your program terminates. After the END trap message, CID isssues a ? prompt. In response to the ? prompt, you can display program variables as they exist at the time of termination, reexecute part or all of your program (see GOTO command described later in this section), or you can terminate the session by entering QUIT. Entering the GO command after the END trap has occurred causes CID to issue an error message because program execution is complete and cannot continue any further.

ABORT Trap

The ABORT trap is useful because it allows you to gain control when program execution terminates abnormally. The values of program variables as they existed at the time of termination can be examined. In some cases it is possible to change program values and reexecute part or all of your program (see GOTO command described later in this section).

The general format of the ABORT trap message is as follows:

```
*T #18, ABORT error message IN L.n
```

where error message indicates the reason the program aborted and n is the line number in which program execution terminated. CID permanently assigns the number 18 to the ABORT trap.

The ABORT trap can also occur when the execution time limit is exceeded. Under NOS/BE, the trap occurs immediately after the time limit is exceeded. Under NOS, the operating system first gains control. You can then direct the operating system to continue or to stop execution. If you direct the operating system to continue execution, the program resumes execution and the ABORT trap does not occur. However, if you direct the operating system to stop execution, CID gains control and the ABORT trap occurs.

Deciding whether or not to continue execution depends on the reason the time limit was exceeded. If your program is executing an infinite loop, you want execution to stop. However, if your program simply requires more time to execute, you want execution to continue. If you are not sure about whether to continue or stop execution, it is usually best to stop execution and consult your program listing to see if your program has an infinite loop.

When a time limit ABORT trap occurs under either operating system, you are given a small amount of additional time to execute CID commands; if this time is exceeded, the debug session is terminated.

When the ABORT trap occurs, the number of the line in which execution stopped is displayed in the trap message. You can then analyze your program listing to find the cause of abnormal termination. For example, in the case of a time limit ABORT trap, you could look for an infinite loop in the area where the time limit occurred. Sometimes it is useful to initiate another debug session and set breakpoints to monitor program values before the time limit is reached.

To illustrate how the ABORT trap works, a program containing an error is executed under CID control. The source listing and session log are shown in figure 3-6. The statement LET C=(A+B)/(A-B) results in division by zero. When division by zero is attempted on line 120, CID immediately gains control and issues a trap message indicating that the program aborted because division by zero was attempted at line 120. The PRINT command is entered to display the contents of variables A, B, C, and D at the time the program aborted. Variables C and D contain the value of zero because the program aborted before they were assigned other values. The QUIT command terminates the session.

INTERRUPT Trap

An INTERRUPT trap gives control to CID when you issue a terminal interrupt.

The general format of the INTERRUPT trap message is as follows:

```
*T #19, INTERRUPT AT L.n
```

where n is the line number at which program execution was suspended. The INTERRUPT trap is permanently assigned the number 19 by CID.

The procedure for issuing a terminal interrupt depends on the terminal type and on the interactive communication system in use. See Interrupt in the Glossary, appendix B.

```
Program ERRBAS:

      00095 REM PROGRAM ERRBAS
      00100 LET A=2.0
      00110 LET B=2.0
      00120 LET C=(A+B)/(A-B)
      00130 LET D=C+1.0
      00140 END


Session Log:

       CYBER INTERACTIVE DEBUG
      ? go<-------------------------------------------- Initiate execution.

       *T #18, ABORT  DIVISION BY ZERO  IN L.120<------- ABORT trap suspends execution at line 120.
      ? print a,b,c,d<--------------------------------- Display values of variables.

        2            2            0            0<------ Variables C and D were unchanged.
      ? quit<------------------------------------------- Terminate session.
```

Figure 3-6. Program ERRBAS and Debug Session Illustrating ABORT Trap

When you enter the appropriate interrupt sequence, the process currently active is interrupted; CID gets control, issues the INTERRUPT trap message, and gives control to you. The INTERRUPT trap can be used to terminate excessive output to the terminal. It can also be used to interrupt a program that you think is looping excessively at some unknown location.

USER-ESTABLISHED TRAPS

In addition to the default traps, CID provides traps that you can establish whenever you have control. The user-established traps described in this guide are the LINE and STORE traps. Up to sixteen user-established traps can be in effect at a given time.

SET,TRAP Command

The traps described in the following paragraphs are established with the SET,TRAP command. This command has the form

```
SET,TRAP,type,scope
```

where type is one of the trap types listed in table 3-1, and scope is one of the notation forms listed in table 3-2. The short form of SET,TRAP is ST.

The scope parameter of the SET,TRAP command specifies the program locations for which the trap is effective. The scope of a trap can be a single location, such as a variable or a BASIC line number, or multiple locations, such as an array or a range of lines.

Not all forms listed in table 3-2 are valid for all CID trap types; valid forms depend on the particular type of trap you set. The forms listed in table 3-2 are valid for the particular trap types described in this guide.

Traps can be established whenever program execution is suspended and CID has issued a ? prompt. If a condition for which you have established a trap does not occur, the program executes normally.

TABLE 3-2. TRAP SCOPE PARAMETERS

Scope	Trap is Set
*	Everywhere
var	For variable var
L.n	At line n
L.m...L.n	Everywhere within the range of lines m thru n (m<n)

LINE Trap

The LINE trap suspends program execution and gives control to CID before execution of each BASIC line within the specified scope occurs. This trap allows you to trace through an executing program and to examine and alter variable values before each statement is executed. The command to set a LINE trap has the form

```
SET,TRAP,LINE,scope
```

where scope has one of the following forms:

*

The trap is set for each line in your program.

L.m...L.n

The trap is set for the range of lines m through n (m<n). Note that no spaces can separate the three periods in this notation.

Some examples of the LINE trap are as follows:

```
SET,TRAP,LINE,*
```

Suspends execution before each executable statement in the program.

SET,TRAP,LINE,L.100...L.140

Suspends execution before each of lines 100 through 140 is executed.

ST,LINE,L.250...L.290

Suspends execution before each of lines 250 through 290 is executed.

To illustrate the LINE trap, the program in figure 3-7 is executed under CID control. The session log is shown in figure 3-8. Program SWAP reads data into variables A and B, and then exchanges their contents using variable C as temporary storage. The first CID command sets a LINE trap at lines 130 through 160. The LINE trap occurs immediately before each executable statement in that range. The PRINT command, entered after each trap occurs, shows the actual exchange of the contents of A and B taking place. The GO command resumes execution after each suspension. Note that both the LINE and the END trap occur at line 160, the last executable statement of the program. This illustrates that more than one trap can occur at the same location.

```
00100 REM PROGRAM SWAP
00110 READ A,B
00120 DATA 3,5
00130 LET C=A
00140 LET A=B
00150 LET B=C
00160 END
```

Figure 3-7. Program SWAP

STORE Trap

The STORE trap suspends execution whenever data is stored in the specified locations. The command to set a STORE trap has the form

SET,TRAP,STORE,var

where var is a simple or subscripted variable in the program.

The STORE trap is useful because it allows you to gain control whenever a specific variable is modified. You can then display the value stored into the variable. A variable is modified whenever a statement in the program is executed in which the variable appears to the left of an equals sign or whenever the variable receives data as a result of an input operation.

Note that while STORE traps can detect stores into any variable or range of variables, BASIC string pointers are sometimes manipulated without affecting the string to which they point. This means that extraneous STORE traps can occur. Therefore, if you use the STORE TRAP for a string variable and a trap is detected, you should inspect the source statement to verify that the string variable has been changed and the trap is not an extraneous one.

Following are some examples of the STORE trap:

SET,TRAP,STORE,A

Suspends execution whenever data is stored in the variable A.

```
 CYBER INTERACTIVE DEBUG
? set,trap,line,l.130...l.160      <-- Set LINE trap at lines 130 through 160.

? go                               <-- Initiate execution.

 *T #1, L AT L.130
? print a;b;c

  3  5  0
? go

 *T #1, L AT L.140
? print a;b;c

  3  5  3
? go                               <-- LINE trap suspends execution at lines 130 through 160. After
                                       each suspension variables A, B, and C are displayed and exe-
 *T #1, L AT L.150                     cution is resumed.
? print a;b;c

  5  5  3
? go

 *T #1, L AT L.160
? print a;b;c

  5  3  3
? go

 *T #17, END IN L.160              <-- END trap occurs at line 160.
? quit
```

Figure 3-8. Debug Session Illustrating LINE Trap

```
SET,TRAP,STORE,B(100)
```

Suspends execution whenever data is stored in array element B(100).

```
ST,STORE,T(3,20)
```

Suspends execution whenever data is stored in array element T(3,20).

To set a STORE trap that is effective for all elements of an array, or for particular elements within an array, use the following ellipsis notation:

```
a(n1)...a(n2)
```

This notation denotes elements n1 through n2 of array a and can be used for one-, two-, or three-dimensional arrays.

The following examples use the ellipsis notation:

```
SET,TRAP,STORE,X(5)...X(12)
```

Suspends execution whenever data is stored in any of the elements X(5) through X(12).

```
SET,TRAP,STORE,C(2,10)...C(3,15)
```

Suspends execution whenever data is stored in any of the elements C(2,10) through C(3,15).

The STORE trap can be helpful in debugging a long program in which a variable is being inadvertently changed at an unknown location.

One significant disadvantage of the STORE trap is that it requires interpret mode execution. (See Interpret Mode following this discussion.) After you set a STORE trap, CID displays the message:

```
INTERPRET MODE TURNED ON
```

Interpret mode greatly increases the execution time a program requires. For this reason, you should not set a STORE trap until you reach a point in a debug session where you want the trap to be effective. When you reach a point in the session where the trap is no longer needed, you should clear it (see Clearing Traps).

To illustrate the STORE trap, the program in figure 3-9 was executed under CID control to produce the debug session in figure 3-10. The STORE trap is set so that CID gets control whenever data is stored into array A. Execution is subsequently suspended on each pass through the loop when the value of variable C is stored into each of the four elements of the array A. Note that because BASIC stores array elements in row order, A(1,1)+1 corresponds to A(1,2), A(1,1)+2 corresponds to A(2,1), and A(1,1)+3 corresponds to A(2,2).

```
00100 REM PROGRAM ARRFILL
00110 OPTION BASE 1
00120 DIM A(2,2)
00130 LET C=1
00140 FOR I=1 TO 2
00150 FOR J=1 TO 2
00160 LET A(I,J)=C
00170 LET C=C+1
00180 NEXT J
00190 NEXT I
00200 END
```

Figure 3-9. Program ARRFILL

LISTING TRAPS

To display a list of traps defined for a debug session, enter one of the following forms of the LIST,TRAP command:

```
LIST,TRAP,*
```

Lists the type, number, and location of all currently defined traps.

```
LIST,TRAP,type,*
```

Lists all traps of the specified type that are currently defined.

```
LIST,TRAP,type,scope
```

Lists the trap identified by the specified type and scope.

```
LIST,TRAP,#n1,#n2,...,#nm
```

Lists all traps identified by the specified number.

The short form of LIST,TRAP is LT.

LIST,TRAP output has the following form:

```
T #n = type scope
```

where n is the trap number assigned by CID, type is the trap type as listed in table 3-1, and scope is the location of the trap in the form specified in the SET,TRAP command.

If no traps exist when LIST,TRAP is entered, CID displays the message:

```
NO TRAPS
```

Figure 3-11 shows a debug session for the program shown in figure 3-7 in which some traps are established and then listed.

```
 CYBER INTERACTIVE DEBUG
? set,trap,store,a(1,1)...a(2,2)  <-- Set store trap for array A.

 INTERPRET MODE TURNED ON
? go

 *T #1, STORE INTO A(1,1) IN L.160
? print c;a(1,1)

  1  1
? go

 *T #1, STORE INTO A(1,1)+1 IN L.160
? print c;a(1,2)

  2  2
? go                              <-- STORE trap suspends execution each time a value is stored
                                      in array A.  Note that A(1,1) corresponds to the first word
                                      of the array, A(1,1)+1 to the second word, and so forth.
 *T #1, STORE INTO A(1,1)+2 IN L.160
? print c;a(2,1)

  3  3
? go

 *T #1, STORE INTO A(1,1)+3 IN L.160
? print c;a(2,2)

  4  4
? go

 *T #17, END IN L.200
? quit
```

Figure 3-10. Debug Session Illustrating STORE Trap

```
 CYBER INTERACTIVE DEBUG
? set,trap,line,l.130...l.160  <-- Set LINE trap.

? set,trap,store,a             <-- Set STORE trap for variable A.

 INTERPRET MODE TURNED ON
? go                           <-- Initiate execution.

 *T #2, STORE INTO A IN L.110
? list,trap,*                  <-- Display trap information.

 T #1 = LINE L.130...L.160,   T #2 = STORE A
?
```

Figure 3-11. Debug Session Illustrating LIST,TRAP Command

CLEARING TRAPS

When a user-defined trap is no longer needed in the debug session, you can clear it by using the CLEAR,TRAP command. This command has the following forms:

CLEAR,TRAP,*

Removes all user-defined traps in your program.

CLEAR,TRAP,type,*

Removes all traps of the specified type.

CLEAR,TRAP,type,scope

Removes trap identified by the specified type and scope.

CLEAR,TRAP,#n1,#n2,...,#nm

Removes the traps identified by the specified numbers.

The short form of CLEAR,TRAP is CT.

The type parameter can be any of the types listed in table 3-1 except for the default INTERRUPT, END, and ABORT traps which cannot be cleared.

Following are some examples of the CLEAR,TRAP command:

```
CLEAR,TRAP,LINE,*
```

Clears all LINE traps.

```
CLEAR,TRAP,STORE,*
```

Clears all STORE traps.

```
CLEAR,TRAP,#2,#4,#5
```

Clears the traps identified by trap numbers 2, 4, and 5.

```
CT,*
```

Clears all traps.

A debug session using the CLEAR,TRAP command is illustrated in figure 3-12 (using program ARRFILL shown in figure 3-9). The CLEAR,TRAP command is issued after the third pass through the loop, allowing the program to run to completion without interruption. Note that the END trap is not removed by the CLEAR,TRAP command.

INTERPRET MODE

The STORE trap requires a mode of execution called interpret mode. In interpret mode, each machine instruction is simulated by CID. Interpret mode is automatically activated when a STORE trap is set; it remains on until the trap is cleared by a CLEAR,TRAP command or until explicitly turned off. CID indicates interpret mode by issuing the message:

```
INTERPRET MODE TURNED ON
```

Execution in interpret mode is much more time-consuming than normal execution. For this reason, you should use STORE traps sparingly.

You can reduce the amount of execution time required for interpret mode by turning interpret mode off while executing portions of a program not currently being debugged. The commands to turn off interpret mode are:

```
SET,INTERPRET,OFF
```

or

```
CLEAR,INTERPRET
```

Traps requiring interpret mode become inoperative when interpret mode is turned off. You can reactivate them with the command:

```
SET,INTERPRET,ON
```

To illustrate the SET,INTERPRET command, the program shown in figure 3-13 is executed in debug mode to produce the debug session shown in figure 3-14. In this example, a STORE trap, which activates interpret mode, is established for variable K in the main portion of the program. Interpret mode is then turned off while subroutine SETB is executing. To accomplish this, breakpoints are set at the beginning and at the end of the subroutine. When execution is suspended at the first breakpoint, interpret mode is turned off; when execution is suspended at the second breakpoint, interpret mode is turned back on, reactivating the STORE trap.

This method of turning off interpret mode is rather cumbersome since the SET,INTERPRET commands must be entered on each pass through the subroutine. A better method is to include the SET,INTERPRET commands in a command sequence so they can be executed automatically. Command sequences are described in section 5.

```
 CYBER INTERACTIVE DEBUG
? set,trap,store,a(1,1)...a(2,2)      <-- Set STORE trap for array A.

 INTERPRET MODE TURNED ON
? go                                  <-- Initiate execution.

 *T #1, STORE INTO A(1,1) IN L.160
? go
 *T #1, STORE INTO A(1,1)+1 IN L.160  <-- STORE trap suspends execution on each store into array A.
? go
 *T #1, STORE INTO A(1,1)+2 IN L.160
? clear,trap,store,*                  <-- Remove STORE trap.

 INTERPRET MODE TURNED OFF
? list,trap,*                         <-- Display trap information.

 NO TRAPS
? go                                  <-- Resume execution.

 *T #17, END IN L.200                 <-- Program runs to completion.
? quit
```

Figure 3-12. Debug Session Illustrating CLEAR,TRAP Command

```
00100 REM PROGRAM ARRYB
00110 OPTION BASE 1
00120 DIM B(5)
00130 LET N=5
00140 LET K=1
00150 GOSUB 200
00160 LET K=2
00170 GOSUB 200
00180 STOP
00190 REM SUBROUTINE SETB
00200 IF K=2 THEN 250
00210 FOR I=1 TO N
00220 LET B(I)=1
00230 NEXT I
00240 RETURN
00250 FOR I=1 TO N
00260 LET B(I)=-1
00270 NEXT I
00280 RETURN
00290 END
```

Figure 3-13. Program ARRYB

SUMMARY OF BREAKPOINT AND TRAP CHARACTERISTICS

The following is a summary of breakpoint and trap information presented in this section:

Breakpoints and traps can be set, cleared, or listed any time CID has control and has prompted you for input.

Only one breakpoint can be established at a single statement; however, a single breakpoint and one trap of each type can be set to occur at a single statement.

Breakpoints and traps exist for the duration of the debug session unless removed by a CLEAR command or inhibited by the SET,INTERPRET,OFF command before the session is terminated.

The frequency parameters of the SET,BREAKPOINT command can be used to avoid suspending execution on each pass through a loop.

CID automatically establishes END, ABORT, and INTERRUPT traps so that you receive control on any program termination, even if you have not explicitly established any breakpoints or traps.

Breakpoints suspend execution before the statement at the breakpoint location is executed. The point in the execution of a statement at which a trap suspends execution depends on the trap type. When execution is resumed, the statement at the breakpoint or trap location is executed in a normal manner.

The STORE trap activates INTERPRET mode, which increases execution time. Execution time can be reduced by specifying the SET,INTERPRET,OFF command when executing portions of the program already debugged.

```
 CYBER INTERACTIVE DEBUG
? set,trap,store,k <-------------- Set STORE trap for variable K.

 INTERPRET MODE TURNED ON
? set,breakpoint,l.200 <---------- Set breakpoint at first executable statement of subroutine SETB.

? set,breakpoint,l.240 )
                       ) <-------- Set breakpoints at the RETURN statements in SETB.
? set,breakpoint,l.280 )

? go

 *T #1, STORE INTO K IN L.140 <--- STORE trap at line 140.
? go

 *B #1, AT L.200 <---------------- Breakpoint suspends execution at line 200.
? set,interpret,off <------------- Turn off interpret mode. (STORE trap inhibited.)

? go

 *B #2, AT L.240 <---------------- Breakpoint suspends execution at line 240.
? set,interpret,on <-------------- Turn on interpret mode. (STORE trap reactivated.)

? go

 *T #1, STORE INTO K IN L.160 <--- STORE trap at line 160.
? print k

  2
? quit
```

Figure 3-14. Debug Session Illustrating SET,INTERPRET Command

DISPLAYING PROGRAM VARIABLES

When program execution is suspended and CID has prompted you for input, you can enter commands to display the values of program variables as they existed at the time of suspension. This discussion includes the most useful forms of the display commands.

CID provides four commands for displaying the values of program variables: the PRINT command, the MAT PRINT command, the LIST,VALUES command, and the DISPLAY command. These commands are summarized in table 3-3.

PRINT COMMAND

The PRINT command, introduced in section 2, is the most useful of the display commands for the BASIC programmer. This command is similar in format and function to the BASIC list-directed PRINT statement. The format is:

```
PRINT output-list
```

The output-list elements must be separated by commas or semicolons and can consist of any of the following:

Simple and subscripted variables

Numeric and string constants

BASIC expressions not involving exponentiation or functions

Variables used in the output-list must exist in the program. Expressions cannot contain references to functions or to the exponentiation operator. CID does not support partial print lines. The trailing comma or semicolon is ignored in CID. Images, PRINT USING statements, and file ordinals cannot be used in CID.

Figure 3-15 shows a program that reads five names and ages, then sorts the names in alphabetical order, keeping the age with the corresponding name.

```
00100 REM PROGRAM SORT
00110 OPTION BASE 1
00120 DIM N$(5)
00130 DIM A(5)
00140 PRINT "INPUT 5 NAMES";
00150 MAT INPUT N$
00160 PRINT "INPUT 5 AGES";
00170 MAT INPUT A
00180 FOR I=1 TO 4
00190 FOR J=I+1 TO 5
00200 IF N$(I)>N$(J) THEN GOSUB 00250
00210 NEXT J
00220 NEXT I
00230 STOP
00240 REM SUBROUTINE SWAP
00250 LET T$=N$(I)
00260 LET N$(I)=N$(J)
00270 LET N$(J)=T$
00280 LET S=A(I)
00290 LET A(I)=A(J)
00300 LET A(J)=S
00310 RETURN
00320 END
```

Figure 3-15. Program SORT

A debug session illustrating the PRINT command is shown in figure 3-16. Note that in displaying the elements of N$ you must specify the subscript. To display the entire array N$ using the PRINT command, you must specify each element of the array separately. Note also the substring reference that is used for the last PRINT command in the session.

MAT PRINT COMMAND

The MAT PRINT command is similar to the BASIC MAT PRINT statement. It prints complete one-, two-, or three-dimensional arrays. (Note that the BASIC MAT PRINT statement does not support three-dimensional arrays.) The format for this command is as follows:

```
MAT PRINT array-list
```

where array-list is a list of one or more one-, two-, or three-dimensional arrays separated by commas or semicolons.

TABLE 3-3. DISPLAY COMMANDS

Command	Description	Formatting	Scope
PRINT	Displays contents of specified variables	Automatic according to variable type	Home program only
MAT PRINT	Displays contents of specified arrays	Automatic according to variable type	Home program only
LIST,VALUES	Lists alphabetically all variable names and values within specified scope	Automatic according to variable type	Specified program unit; entire program if none specified
DISPLAY	Displays contents of specified variable	User-specified; default is variable type	Default is home program; variables can be qualified for other than home program

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.200 <-------------------- Set breakpoint at line 200.

? go <-------------------------------------- Initiate execution.

INPUT 5 NAMES ? van,dana,sheila,betty,rick  <-- Input data.
INPUT 5 AGES? 38,62,25,49,57

 *B #1, AT L.200 <--------------------------- Breakpoint detected at line 200.
? print i;j <------------------------------- Display variables I and J.

  1  2
? go <-------------------------------------- Resume execution.

 *B #1, AT L.200
? print i;j

  1  3
? go

 *B #1, AT L.200
? print i;j

  1  4
? print n$(1);a(1),n$(4);a(4) <------------- Display names and ages of the two elements to be compared.
 DANA 62        BETTY 49
? clear,breakpoint,#1 <--------------------- Clear breakpoint number 1.

? go

 *T #17, END IN L.230
? print i;j;t$(1:2);s <--------------------- Display final values of variables I, J, T$, and S.

  5  6 VA 38
? quit
```

Figure 3-16. Debug Session Illustrating PRINT Command

Following are some examples of the MAT PRINT command:

```
MAT PRINT A,B
```

Prints the arrays A and B.

```
MAT PRINT X1$
```

Prints the string array X1$.

Arrays listed in the array list must exist in the BASIC program. Elements of the array are printed in row order (the rightmost subscript changes most rapidly) with spacing between items controlled by the comma or semicolons as it is for the PRINT command. A blank line is output after each row and an extra blank line is output between arrays. For a three-dimensional array, each plane is printed in row order. Two blank lines separate one plane from another. An extra blank line is output between arrays.

Assuming the base set in the OPTION statement is one, the MAT PRINT command prints a 2x2 array A as follows:

```
A(1,1)  A(1,2)
A(2,1)  A(2,2)
```

A 2x2x2 array A is printed as follows:

```
A(1,1,1)  A(1,1,2)
A(1,2,1)  A(1,2,2)

A(2,1,1)  A(2,1,2)
A(2,2,1)  A(2,2,2)
```

Figure 3-17 shows a debug session illustrating the MAT PRINT command. The program in figure 3-15 was executed under CID control to produce the session log. A breakpoint is set at line 180 and program execution is begun. When the breakpoint is detected, the MAT PRINT command is entered to display unsorted arrays N$ and A. The breakpoint is cleared, execution is resumed, and the program runs to completion. After the END trap occurs, the MAT PRINT command is again entered to display the sorted arrays N$ and A.

LIST,VALUES COMMAND

The LIST,VALUES command alphabetically lists all variables defined in the source program and the current value of each. This command automatically formats the variables according to the variable type as declared in the source program. The format of the command is as follows:

```
LIST,VALUES
```

The short form of LIST,VALUES is LV.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.180 <-------------------------- Set breakpoint at line 180.

? go
INPUT 5 NAMES ? van,dana,sheila,betty,rick  }
INPUT 5 AGES? 38,62,25,49,57                } <---- Input data.
 *B #1, AT L.180 <-------------------------------- Breakpoint detected at line 180.
? mat print n$,a <------------------------------- Display the unsorted arrays.
 VAN            DANA           SHEILA         BETTY          RICK

  38             62             25             49             57


? clear,breakpoint,* <--------------------------- Clear all breakpoints.

? go <------------------------------------------- Resume execution.

 *T #17, END IN L.230
? mat print n$,a <------------------------------- Display the sorted arrays.
 BETTY          DANA           RICK           SHEILA         VAN

  49             62             57             25             38


? quit
```

Figure 3-17. Debug Session Illustrating MAT PRINT Command

The LIST,VALUES command provides a formatted snapshot of the values of program variables; however, it can produce excessive output, particularly if the program contains large arrays. You can either send the output to an auxiliary file (described in section 4) or use an alternate command to display selected values. Use the PRINT, MAT PRINT, or DISPLAY command to avoid large amounts of output.

Because the LIST,VALUES command displays all program variables, it is slow and can add substantially to the execution time of a debug session, particularly for programs with many variables. It is well worth the additional time in many cases, but alternate commands should always be considered.

To illustrate the LIST,VALUES command, the program shown in figure 3-15 is executed under CID control to produce the debug session in figure 3-18. A breakpoint is set at line 180 so that execution is suspended before any sorting occurs. When the breakpoint is detected, the LIST,VALUES command is entered. The program values are formatted according to their type. The breakpoint is cleared, the GO command is entered, and execution resumes. When the program terminates, the LIST,VALUES command is entered again.

DISPLAY COMMAND

The DISPLAY command displays the contents of specified variables. The DISPLAY cannot be used to display string variables. If you try to display a string variable with the DISPLAY command, the contents of the string pointer, not the value of the string, is printed.

You should use the PRINT command in most cases because it provides for automatic formatting of variables and is more familiar. Also, while the PRINT command allows you to specify a list of variables, a separate DISPLAY command must be used for each variable. The DISPLAY command, however, offers the following advantages:

- DISPLAY allows you to specify the format of each variable, whereas PRINT performs automatic formatting. In most cases, automatic formatting is more convenient. However, in situations where you want to display the value of a variable in a format other than its declared or implicit format, you must use the DISPLAY command.

- DISPLAY is the only command that can display the values of debug variables (described later in this section).

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.180 <-------------------------- Set breakpoint at line 180.

? go <-------------------------------------------- Initiate execution.
INPUT 5 NAMES ? van,dana,sheila,betty,rick  }
INPUT 5 AGES? 38,62,25,49,57                } <--- Input data.
 *B #1, AT L.180 <-------------------------------- Breakpoint detected at line 180.
? list,values <----------------------------------- Display all program variables
                                                    and their values.
 P.SORTBAS
 A(1) =  38,   A(2) =  62,   A(3) =  25,   A(4) =  49,   A(5) =  57
 I =  1,   J =  0,   N$(1) = "VAN",   N$(2) = "DANA"
 N$(3) = "SHEILA",   N$(4) = "BETTY",   N$(5) = "RICK",   S =  0
 T$ = ""
? clear,breakpoint,l.180 <------------------------ Clear breakpoint at line 180.

? go <-------------------------------------------- Resume execution.

 *T #17, END IN L.230
? list,values <----------------------------------- Display all program variables
                                                    and their final values.
 P.SORTBAS
 A(1) =  49,   A(2) =  62,   A(3) =  57,   A(4) =  25,   A(5) =  38
 I =  5,   J =  6,   N$(1) = "BETTY",   N$(2) = "DANA",   N$(3) = "RICK"
 N$(4) = "SHEILA",   N$(5) = "VAN",   S =  38,   T$ = "VAN"
? quit
```

Figure 3-18. Debug Session Illustrating LIST,VALUES Command

The DISPLAY command has the format

```
DISPLAY,variable,format
```

where variable is a simple or subscripted variable in the source program, and format is an optional format indicator whose valid values are as follows:

F	Floating-point (numeric base 10)
I	Integer
C	Character
O	Octal

The default format is the variable type as declared in the program. The short form of DISPLAY is D.

Because the DISPLAY command automatically formats variables, it is necessary to specify the format parameter only when you want to display a variable in a format other than that declared in the BASIC program.

Some examples of the DISPLAY command are as follows:

```
DISPLAY,Z
```

Displays the variable Z in default format.

```
DISPLAY,X(3,7),O
```

Displays element X(3,7) in octal format.

The DISPLAY command displays only the first word of an array when an array name is specified. To display successive elements within an array, specify the first and last words separated by three periods (ellipsis notation), as in the following example:

```
DISPLAY,A(5)...A(10)
```

This statement displays elements 5 through 10 of array A in default format.

ALTERING PROGRAM VALUES (LET COMMAND)

CID provides a command that allows you to alter the values of program variables. The LET command is identical in form and function to the BASIC LET statement. This command allows you to make corrections to your program as execution proceeds, eliminating the need for recompiling each time an error is discovered. The LET command has the form

```
LET variable=expression
```

where variable is a simple or subscripted variable, and expression is any valid BASIC arithmetic expression not involving functions or exponentiation. The LET command functions exactly as in BASIC: the expression is evaluated and its value is assigned to the variable on the left of the equal sign; the previous contents of the receiving variable are destroyed.

The variables referenced in the LET command must exist in the BASIC program being debugged. Multiple assignments, references to functions, and use of the exponentiation operator are not allowed; all other arithmetic operators and the string concatenation operator can be used in the expressions.

You can enter the LET command whenever CID has prompted you for input. For example, if program execution is suspended and you have detected a variable that has an incorrect or illegal value, you can use the LET command to assign a new value to the variable. When you resume execution of the program, the new value is used in subsequent computations involving the altered variable.

Changes made with the LET command do not exist beyond the end of the debug session. When a program is reexecuted, either in debug mode or in normal mode, all program variables have the values defined in the original compiled version.

Following are some examples of the LET command:

```
LET A=B
```

Replaces the current value of A with the current value of B.

```
LET M=N+I-1
```

Evaluates the expression using the current value of N and I and assigns the value to M.

```
LET B(I,J)=A(I,J)*A(I,J)+4.5
```

Evaluates the expression using the current value A(I,J) and assigns the value to B(I,J).

```
LET D$(3:6)="DEFG"
```

Assigns the character expression DEFG to D$(3:6).

Figure 3-19 shows a program and debug session that uses the LET command. The program calculates the average of 10 numbers. The program contains an error: the statement LET A=S*10.0 should be LET A=S/10.0.

To enable the program to execute correctly, a breakpoint is set at line 180. When execution is suspended at this location, the program has already calculated an incorrect value for A. The LET command is then entered to calculate the correct value of A. The new value is used in the subsequent PRINT statement when execution is resumed. The erroneous statement must be replaced by the programmer in the corrected version of the source program.

ALTERING PROGRAM EXECUTION (GOTO COMMAND)

CID provides a command that alters the normal flow of program execution. The GOTO command resumes execution of the program at a specified line number. This command is in the same format as the BASIC statement GOTO. The command format is

```
GOTO n
```

where n is the line where execution is to be continued.

For example,

```
GOTO 150
```

causes program execution to continue at line number 150.

When the GOTO command is executed, program execution will begin at the specified line number and continue until the program reaches a breakpoint or trap, or until the program terminates.

The GOTO command should not be used to initiate execution at the beginning of a debug session because program initialization will not take place. Also, care should be taken when you enter this command because it changes the flow of program execution.

Figure 3-20, which uses the program shown in figure 3-15, shows a debug session illustrating the GOTO command. A breakpoint is set at line 160 so that program execution is suspended before the list of ages is entered. After the breakpoint is detected, the GOTO command is used to resume execution at the line immediately following the input line for the ages. When the program has terminated, the sorted arrays N$ and A are diplayed. Note that all the ages are zero because the line requesting the list of ages was bypassed.

DISPLAYING CID AND PROGRAM STATUS INFORMATION

The following paragraphs describe some CID features and commands that allow you to obtain various kinds of information about the current debug session. These features include:

Debug variables that contain useful information about the current session; the values of these variables can be displayed at the terminal.

LIST commands that can display such things as breakpoint and trap information, and the current status of your program.

DEBUG VARIABLES

CID provides variables that contain information about the current status of the debug session and the executing program. You can display the contents of debug variables whenever you have control. CID updates these variables; you cannot alter their contents directly.

Although many debug variables are primarily intended for use by assembly language programmers, some of them can provide information useful to BASIC programmers. Those variables that are most useful to BASIC programmers are listed in table 3-4. (Home program is discussed in section 4 and groups in section 5.) See the CYBER Interactive Debug reference manual for a description of the other debug variables not listed in the table.

TABLE 3-4. DEBUG VARIABLES

Variable	Description
#LINE	Number of BASIC line executing at time of suspension
#PC	Previous contents; on STORE trap, #PC contains the value previously stored in the trapped variable
#HOME	Home program name
#BP	Number of existing breakpoints
#TP	Number of existing traps
#GP	Number of existing groups

```
Program AVGBAS:

      00100 REM PROGRAM AVGBAS
      00110 DIM X(9)
      00120 LET S=0.0
      00130 MAT READ X
      00140 FOR I=0 TO 9
      00150 LET S=S+X(I)
      00160 NEXT I
      00170 LET A=S*10.0
      00180 PRINT "THE NUMBERS ARE AS FOLLOWS: "
      00190 MAT PRINT X
      00200 PRINT "MEAN: ";A
      00210 DATA 5.2,3.4,9.6,7.8,2.3,1.7,6.9,4.5,13.3,8.1
      00220 END


Session Log:

       CYBER INTERACTIVE DEBUG
      ? set,breakpoint,l.180 <------- Set breakpoint to suspend execution at line 180.

      ? go

       *B #1, AT L.180 <------------- Execution suspended.
      ? print a <------------------- Oisplay value of A.

        628.
      ? let a=s/10.0 <-------------- Calculate correct value for A.

      ? print a <------------------- Display new value of A.

        6.28
      ? go <------------------------ Resume execution.

      THE NUMBERS ARE AS FOLLOWS:
       5.2

       3.4

       9.6

       7.8

       2.3

       1.7                         <-- Program prints data used and new value of A.

       6.9

       4.5

       13.3

       8.1


      MEAN:  6.28
       *T #17, END IN L.220
      ? quit
```

Figure 3-19. Program AVGBAS and Debug Session Illustrating LET Command

To display the contents of a debug variable, you must use the DISPLAY command; debug variables cannot be displayed with the PRINT command or LIST,VALUES command. All variables except #PC are automatically displayed in the appropriate format. Because #PC contains a numeric value, you should specify the F format on the DISPLAY command. Octal format is the default.

The #LINE variable contains the number of the BASIC source line that was executing at the time of suspension. The output of the DISPLAY,#LINE command is P.name L.n, where name is the home program name and n is the line number. CID normally prints this for you when a breakpoint or trap occurs, but you might wish to display the value yourself at times, especially when using command sequences (described in section 5).

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.160 <-------------------------- Set breakpoint at line 160.

? go <-------------------------------------------- Initiate execution.

INPUT 5 NAMES ? van,dana,sheila,betty,rick <------ Input names.
 *B #1, AT L.160 <--------------------------------- Breakpoint detected at line 160.
? goto 180 <-------------------------------------- Resume execution at line 180.

 *T #17, END IN L.230 <---------------------------- Program terminates.
? mat print n$,a <-------------------------------- Display the contents of arrays N$ and A.

 BETTY          DANA          RICK          SHEILA        VAN

  0              0             0             0             0

? quit
```

Figure 3-20. Debug Session Illustrating GOTO Command

The #PC variable can be displayed after execution has been suspended by a STORE trap. #PC contains the previous contents of the variable and can be displayed only when a STORE trap has occured. Note that #PC is not useful when a STORE trap has occured on a string variable.

The #BP and #TP variables contain the numbers of breakpoints and traps, respectively, that are defined for the current debug session. These variables are especially useful for longer, more complex debug sessions.

A debug session using debug variables is illustrated in figure 3-21. The program executed to produce this session is shown in figure 3-3. In this example, a breakpoint and a STORE trap are defined. While execution is suspended, the DISPLAY command is used to display the values of various debug variables. Note that when #PC is displayed, the F option is specified so that the value is displayed in floating point format.

LIST COMMANDS

The LIST commands allow you to list various types of information relevant to the current debug session or to your program. The LIST commands are summarized in table 3-5.

The LIST commands are particularly useful with longer debug sessions in which you are constantly changing the status of the session. For example, you can initially set some breakpoints or traps, clear some or all of them later in the session, and set new ones. With the LIST commands you can keep track of this and other CID information. The LIST,BREAKPOINT and LIST,TRAP commands are described earlier in this section. The LIST,MAP command is described in section 4 and the LIST,GROUP command in section 5.

TABLE 3-5. LIST COMMANDS

Command	Description
LIST,BREAKPOINT	Lists breakpoint information
LIST,TRAP	Lists trap information
LIST,GROUP	Lists command group information
LIST,MAP	Lists load map information
LIST,STATUS	Lists information about current status of debug session
LIST,VALUES	Lists names and values of user-defined variables

Some of the LIST commands can produce a large volume of output. It is possible to prevent this output from appearing at the terminal by writing it to a separate file that can then be printed. The commands to accomplish this are described in section 4 under Control of CID Output.

LIST,STATUS Command

The LIST,STATUS command displays a brief summary of the status of a debug session as it exists at the time the command is issued. This command has the form:

```
LIST,STATUS
```

The short form of LIST,STATUS is LS.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.190

? go

 *B #1, AT L.190
? set,trap,store,a

 INTERPRET MODE TURNED ON
? display,#bp <------------------- Display current number of breakpoints.

 #BP = 1
? display,#tp <------------------- Display current number of traps.

 #TP = 1
? go

 *T #1, STORE INTO A IN L.230
? display,#line <----------------- Display number of line where execution suspended.

 #LINE = P.TRIANGL_L.230
? display,#pc,f <----------------- Display previous contents of changed variable in floating point format.

 #PC = 0.0 <---------------------- Variable A had no previous value.
? clear,breakpoint,#1

? go

THE AREA OF THE TRIANGLE IS  2 <-- Program output.
 *T #1, STORE INTO A IN L.230
? display,#pc,f <----------------- Display previous contents of changed variable in floating point format.

 #PC = 2.0
?
```

Figure 3-21. Debug Session Illustrating Debug Variables

Information displayed by the LIST,STATUS command includes:

- Home program name
- Number of breakpoints currently defined
- Number of traps currently defined
- Number of groups currently defined
- Status of veto mode (on or off)
- Status of interpret mode (on or off)
- Current output options which specify the type of CID output sent to the terminal
- Current auxiliary file options which define an auxiliary output file and specify the type of output to be sent to the auxiliary output file

Figure 3-22 illustrates the use of the LIST,STATUS command. The command is issued near the beginning of the debug session.

SAMPLE DEBUG SESSION

The following paragraphs present some examples of interactive debugging using the commands described in this section.

The program entitled CORRBSC reads pairs of numbers and calculates the correlation coefficient of the numbers. The source listing is shown in figure 3-23.

The correlation coefficient is a means of measuring the degree of statistical correlation between two sets of numbers. The formula for the correlation coefficient is shown in figure 3-24.

The correlation coefficient can have any value between -1 and 1. A coefficient with a magnitude close to 1 indicates close correlation.

The program in figure 3-23 contains a number of errors. The program compiles successfully, but does not run to completion.

To execute the program, some test cases are required. If possible, test cases for which results are known should be included. In the example in figure 3-23, the first test case consists of pairs of equal numbers; if the program is correct, it should calculate a correlation coefficient of 1.0.

The first debug session and the first set of input data are shown in figure 3-25. At first no breakpoints or traps are set; GO is entered to initiate program execution. The program is allowed to execute until termination, at which time the ABORT trap gives control to you. The trap message indicates that a subscript error has occurred.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.190

? go

 *B #1, AT L.190
? set,trap,store,a

 INTERPRET MODE TURNED ON
? list,status

 HOME = P.TRIANGL,   1 BREAKPOINTS,   1 TRAPS,   NO GROUPS,   VETO OFF
 INTERPRET ON,   OUT OPTIONS = I W E D,   AUXILIARY CLEAR
?
```

Figure 3-22. Debug Session Illustrating LIST,STATUS Command

```
00100 REM PROGRAM CORRBSC
00110 REM CORRBSC CALCULATES A CORRELATION COEFFICIENT
00120 OPTION BASE 1
00130 DIM X(5),Y(5)
00140 REM INITIALIZATION
00150 LET S1=0
00160 LET S2=0
00170 LET S3=0
00180 LET S4=0
00190 LET S5=0
00200 LET M=1
00210 REM READ IN NUMBERS TO BE CORRELATED
00220 FILE #1="CORRFIL"
00230 IF END #1 GOTO 00270
00240 INPUT #1,X(M),Y(M)
00250 LET M=M+1
00260 GOTO 00230
00270 IF M<>0 THEN GOTO 00310
00280 PRINT "EMPTY INPUT FILE"
00290 STOP
00300 REM CALCULATE THE CORRELATION COEFFICIENT
00310 FOR I=1 TO M
00320 LET S1=S1+X(I)
00330 LET S2=S2+Y(I)
00340 LET S3=S3+X(I)^2
00350 LET S4=S4+Y(I)^2
00360 LET S5=X(I)+Y(I)
00370 NEXT I
00380 LET N=(M*S5-S1*S2)^2
00390 LET D=(M*S3-S1^2)*(M*S4-S2^2)
00400 LET R=SQR(N/D)
00410 PRINT "CORRELATION COFFICIENT = ";R
00420 END
```

Figure 3-23. Program CORRBSC Before Debugging

$$r = \frac{n\Sigma xy - \Sigma x\,\Sigma y}{\sqrt{n\Sigma x^2-(\Sigma x)^2}\quad\sqrt{n\Sigma y^2-(\Sigma y)^2}}$$

r	Correlation coefficient
n	Number of pairs to be correlated
x,y	Values to be correlated
Σ	Summation symbol; for example, Σx is the sum of all x values

Figure 3-24. Correlation Coefficient Formula

Commands now can be entered to determine the cause of the error. The PRINT command displays the value of variable M, which is one too large. By analyzing the program listing, it can be seen that there is a logic error in the program which will always cause M to be one greater than the number of data pairs read. Although not shown in the debug session, this also implies that because the program contains no check on the number of records read, an array bounds error can occur if the number of records exceeds the size of the array. The program can be corrected by initializing M to 0 at line 200, switching lines 240 and 250, and inserting a test for the number of data pairs read. However, in order to learn as much as possible from this debug session, the value of M is temporarily corrected, using the LET command, and the session is continued.

```
Input Data:

     1.0  1.0
     5.1  5.1
     100.5  100.5
     10.0  10.0
     7.6  7.6


Session Log:

      CYBER INTERACTIVE DEBUG
     ? go <------------------------------------------------------ Initiate execution.

      *T #18, ABORT  SUBSCRIPT ERROR  IN L.320 <---------------- ABORT trap occurs.
     ? print m <-------------------------------------------------- Print value of variable M.

       6
     ? let m=5 <-------------------------------------------------- Change value of M.

     ? mat print x,y <-------------------------------------------- Print contents of arrays X and Y.

       1            5.1          100.5        10          7.6

       1            5.1          100.5        10          7.6

     ? goto 380 <------------------------------------------------- Continue execution at line 380.

     CORRELATION COFFICIENT =  .426385 <-------------------------- Program prints result.
      *T #17, END IN L.420
     ? quit
```

Figure 3-25. Input Data for First Test Case and Debug Session

Next, the MAT PRINT command is entered to display the contents of arrays X and Y. After this, the GOTO command is entered to continue program execution at line 380. The program terminates normally, but the value of R is incorrect. The debug session is then terminated.

The second session is shown in figure 3-26. (The program in its original form is used for this session.) Breakpoints are set at lines 270, 370, and 400. The breakpoint is set at line 370 to suspend execution on each pass through the loop of lines 310 through 370.

Execution is initiated and the correct value is calculated for M. The display of intermediate values on each pass through the loop indicates a possible error: the value of the variable S5 should be increasing on each pass as more values are added to it. However, the display shows that this value is not increasing. The calculation of S5 in line 360 is incorrect; the correct statement is

```
LET S5=S5+X(I)*Y(I)
```

The debug session can be continued by using the LET command to calculate and insert the correct value of S5. First, execution is resumed to allow the loop to complete. After the last pass through the loop, the correct value is calculated.

The next suspension occurs at line 400. The values of N and D are printed. For R to have a value of 1.0, N and D must be equal, and the PRINT command shows that this is the case. Execution is resumed at line 400 which calculates the final result. The program runs to completion and the session is terminated. The program now appears to execute correctly.

At this point, it is a good idea to incorporate all the accumulated changes into the source program, recompile, and rerun the program with the same data to verify the corrections. The program, however, should not be considered completely debugged until it has been tested on additional sets of input data.

For the next test case, data records are included in which all the X values are equal. The input file and session log are shown in figure 3-27. The program aborts because division by zero is attempted in line 400. By using CID commands to display intermediate values, you can see that variable D, the denominator in the equation at line 400, has the value of 0. CID has helped determine the location of the error, but in order to understand why the error occurred, it is necessary to understand the mathematics of the program.

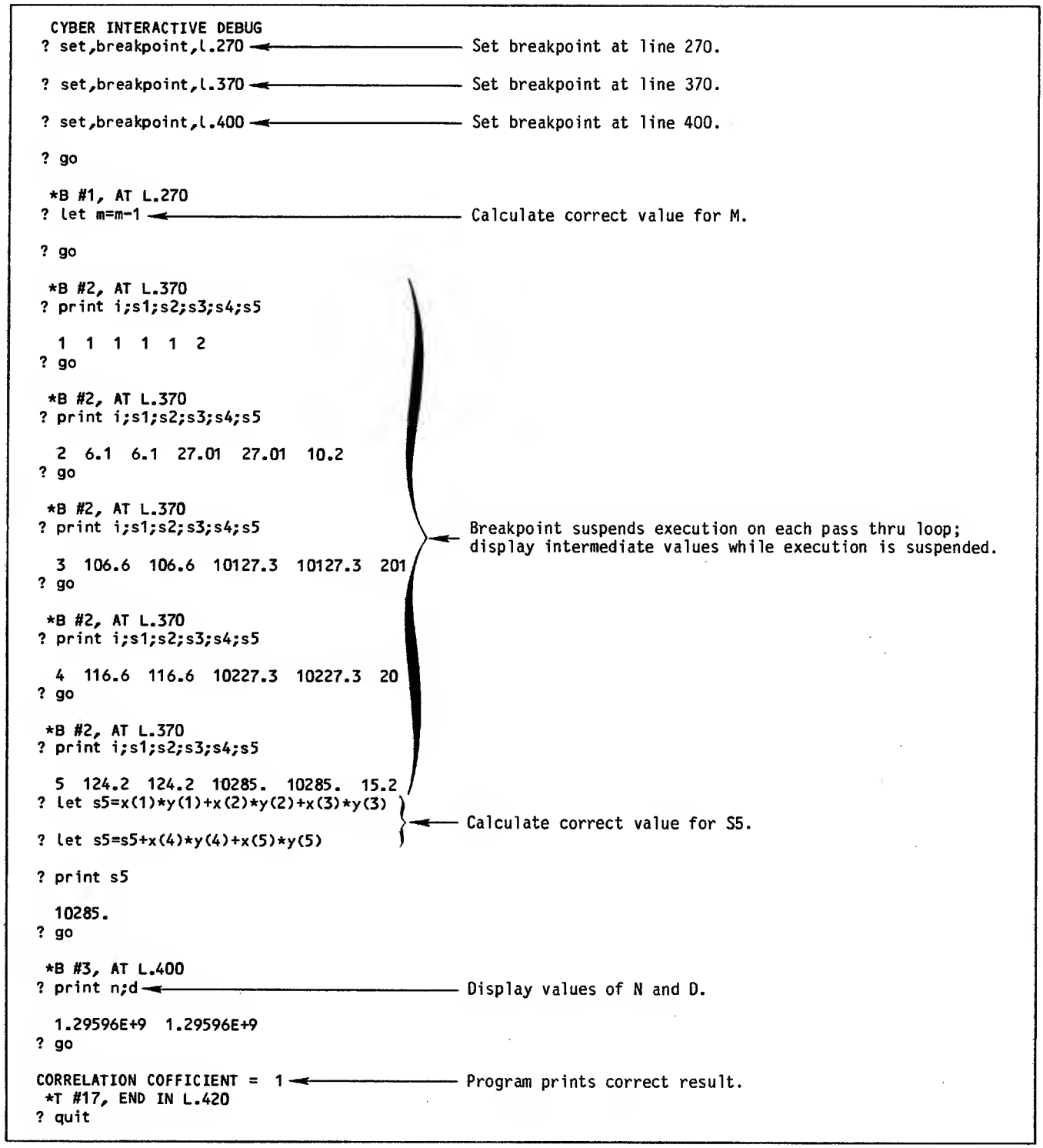

Figure 3-26. Second Debug Session

```
Input Data:

      3.0  1.0
      3.0  5.1
      3.0  100.5
      3.0  10.0
      3.0  7.6


Session Log:

       CYBER INTERACTIVE DEBUG
      ? go <------------------------------------ Initiate execution.

       *T #18, ABORT  DIVISION BY ZERO  IN L.400
      ? print r;n;d <---------------------------- Display values of R, N, and D.

        0  3.05900E+6  0 <-------------------------- Value of 0 for D caused the division by zero error
      ? quit                                       message.
```

Figure 3-27. Input Data for Second Test Case and Debug Session

In the formula for the correlation coefficient, the calculation $n\sum x^2 - (\sum x)^2$ has a value of zero if all the x values are equal. Whenever a division occurs in a program, you should always be alert to the possibility of a zero denominator and include statements testing for that possibility.

To complete the debugging process, two more test cases are run in one debug session: one in which closely correlated data values are input from the file CORFILL, and one in which widely scattered data values are entered using the CID LET command (figure 3-28). Using CID to enter test data is a convenient technique for running test cases, but you must be sure that all variables are initialized correctly before you reexecute part or all of a program.

The results of both tests appear to be correct. In a real situation, the results should be verified whenever possible by comparing them with known results or by performing hand calculations. The final version of CORRBSC, with all corrections included, is shown in figure 3-29.

```
Input Data:

      10.1 10.1
      20.5 20.5
      6.0 6.0
      34.0 32.9
      4.4 4.5


Session Log:

       CYBER INTERACTIVE DEBUG
      ? go <------------------------------------ Initiate execution.

      CORRELATION COFFICIENT =  .999799 <-------- Program prints result.
       *T #17, END IN L.420 <--------------------- Program execution terminates.
      ? set,breakpoint,l.220 )
                             )
      ? goto 150             ) <------------------ Reinitialize program variables, but
                             )                    stop before reading file CORRFIL.
       *B #1, AT L.220       )
      ? let x(1)=0.0         )
                             )
      ? let y(1)=5.0         )
                             )
      ? let x(2)=0.1         ) <------------------ Input new values for X(1), Y(1),
                             )                    X(2), Y(2), X(3) and Y(3).
      ? let y(2)=500         )
                             )
      ? let x(3)=100         )
                             )
      ? let y(3)=0.0         )
```

Figure 3-28. Input Data for Third and Fourth Test Cases and Debug Session (Sheet 1 of 2)

```
? let m=3 <------------------------------------------ Set M so that only the values just
                                                      input will be used to calculate R.
? mat print x,y <------------------------------------ Print the contents of arrays X and Y.

  0           .1          100          34          4.4

  5           500         0            32.9        4.5

? goto 270 <----------------------------------------- Resume program execution.

CORRELATION COFFICIENT =  .506772 <------------------ Program prints result.
 *T #17, END IN L.420
? quit
```

Figure 3-28. Input Data for Third and Fourth Test Cases and Debug Session (Sheet 2 of 2)

```
00100 REM PROGRAM CORRBSC
00110 REM CORRBSC CALCULATES A CORRELATION COEFFICIENT
00120 OPTION BASE 1
00130 DIM X(5),Y(5)
00140 REM INITIALIZATION
00150 LET S1=0
00160 LET S2=0
00170 LET S3=0
00180 LET S4=0
00190 LET S5=0
00200 LET M=0†
00210 REM READ IN NUMBERS TO BE CORRELATED
00220 FILE #1="CORRFIL"
00230 IF END #1 GOTO 00300
00240 LET M=M+1†
00250 IF M>5 THEN GOTO 00280†
00260 INPUT #1,X(M),Y(M)†
00270 GOTO 00230
00280 PRINT "TOO MUCH INPUT. LIMIT IS 5 PAIRS."†
00290 LET M=M-1†
00300 IF M<>0 THEN GOTO 00340
00310 PRINT "EMPTY INPUT FILE"
00320 STOP
00330 REM CALCULATE THE CORRELATION COEFFICIENT
00340 FOR I=1 TO M
00350 LET S1=S1+X(I)
00360 LET S2=S2+Y(I)
00370 LET S3=S3+X(I)^2
00380 LET S4=S4+Y(I)^2
00390 LET S5=S5+X(I)*Y(I)†
00400 NEXT I
00410 LET N=(M*S5-S1*S2)^2
00420 LET D=(M*S3-S1^2)*(M*S4-S2^2)
00430 IF D<>0 THEN 00460†
00440 PRINT "ERROR-ALL X'S ARE EQUAL"†
00450 STOP†
00460 LET R=SQR(N/D)
00470 PRINT "CORRELATION COFFICIENT = ";R
00480 END
```

†Indicates correction.

Figure 3-29. Program CORRBSC With Corrections

This section describes some advanced CYBER Interactive Debug (CID) tools. The commands described in this section allow you to do the following:

Execute your program a few lines at a time; the command is STEP.

Control output created by CID commands; the commands are SET,OUTPUT and SET,AUXILIARY.

Reference FORTRAN subroutines outside your BASIC main program; the command is SET,HOME.

Keep track of the current home program and other program units; the commands are DISPLAY,#HOME and LIST,MAP.

EXECUTING A FEW LINES AT A TIME (STEP COMMAND)

The STEP command initiates or resumes program execution until a specified number of lines have been executed and then gives you control. The format of the STEP command is as follows:

```
STEP,n,LINE,scope
```

where n is the number of lines to be counted before execution is suspended (default is 1) and scope is the address range indicating which lines are to be counted. The short form of STEP is S.

The scope parameter can take one of the following forms:

*

All lines are to be counted (default value).

L.m...L.n

All lines in the specified address range are to be counted (m<n).

If no parameters are specified, the previous STEP command is reexecuted. If no prior STEP command exists for the current debug session, the default values for parameters n and scope are used and, therefore, STEP,1,LINE,* is executed.

When you gain control as a result of a STEP command, CID suspends program execution and issues the following message:

```
*S LINE AT address
```

The address is a reference to the next line to be executed. CID next issues the ? prompt for input. In response, you can enter any CID command.

If a breakpoint or trap suspends program execution before a STEP command has completed its action, the STEP command is discontinued. For example, if you issue the command

```
STEP,5,LINE
```

and a breakpoint occurs after two lines are executed, you gain control at the breakpoint, and the STEP command is discontinued.

If a STEP command suspends program execution at the same time as a breakpoint or trap, the breakpoint or trap takes precedence and the STEP command is considered finished. You must issue another STEP command to continue stepping through your program.

In the following examples, the STEP command shown will be executed as described unless a breakpoint or trap gives you control before the STEP command has completed its action.

```
STEP,6,LINE
```

Execution is suspended after the next six lines in the program have been executed.

```
STEP,1
```

Execution is suspended after the next line in the program is executed.

```
S,1,LINE,L.240...L.380
```

Execution is suspended after line 240 is executed. To continue suspending execution one line at a time for the entire specified scope, enter the command STEP without parameters after each suspension occurs.

The debug session in figure 4-1, produced by executing the program in figure 3-15 (see section 3) in debug mode, illustrates the STEP command. The STEP command is used to allow subroutine SWAP to be executed a few lines at a time. When the STEP command causes execution to be suspended, the names and ages being exchanged are displayed to make sure the swap is executing properly. Note that when the second STEP command, which does not specify any parameters, is issued, STEP,3,LINE is reexecuted.

CONTROL OF CID OUTPUT

The CID commands PRINT, MAT PRINT, LIST, and DISPLAY can generate large amounts of output. As an alternative to displaying all CID output at the terminal, you can define an auxiliary output file and specify certain types of CID output to be written to the file. The commands to control CID output are SET,OUTPUT and SET,AUXILIARY.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.250 <-------- Set breakpoint at line 250.

? go <-------- Initiate execution.

INPUT 5 NAMES ? van,dana,sheila,betty,rick
INPUT 5 AGES? 38,62,25,49,57

 *B #1, AT L.250 <-------- Breakpoint suspends execution at line 250.
? print n$(i),n$(j) <-------- Display names to be swapped.

 VAN            DANA
? step,3,line <-------- Execute three lines and suspend execution.

 *S LINE AT L.280 <-------- STEP suspends execution at line 280.
? print n$(i),n$(j) <-------- Display exchanged names.

 DANA           VAN
? print a(i),a(j) <-------- Display ages to be swapped.

  38             62
? step <-------- STEP,3,LINE is reexecuted.

 *S LINE AT L.310 <-------- STEP suspends execution at line 310.
? print a(i),a(j) <-------- Display exchanged ages.

  62             38
? go <-------- Terminate STEP.

 *B #1, AT L.250
? print n$(i),n$(j),a(i);a(j)

 DANA           BETTY          62  49
? step,6,line <-------- Execute six lines and suspend execution.

 *S LINE AT L.310 <-------- STEP suspends execution at line 310.
? print n$(i),n$(j),a(i);a(j)

 BETTY          DANA           49  62
? quit
```

Figure 4-1. Debug Session Illustrating STEP Command

TYPES OF OUTPUT

For use with the SET,OUTPUT and SET,AUXILIARY commands, CID output is classified according to type, with each type represented by a one-letter code. The output type codes along with their descriptions are listed in table 4-1.

SET,OUTPUT COMMAND

The SET,OUTPUT command specifies the types of output to be displayed at the terminal. The SET,OUTPUT command has the form

```
SET,OUTPUT,type-list
```

where type-list is a list of output type codes as shown in table 4-1. The type codes can be separated by commas or entered without separators. The short form of SET,OUTPUT is SOUT.

Including an output code in the option list of the SET,OUTPUT command causes the associated output type to be displayed at the terminal. Omitting an output code from the option list suppresses the associated output type. Thus, when you specify a SET,OUTPUT command, any output type excluded from the option list is not displayed at the terminal.

TABLE 4-1. CID OUTPUT TYPES

Output Code	Description
E	Error messages
W	Warning messages
D	Debug output produced by execution of CID commands; includes output produced by LIST, PRINT, MAT PRINT, and DISPLAY commands
I	Informative messages; includes trap and breakpoint messages
R	Echo commands read from group and file command sequences when a READ command is executed
B	Echo commands read from trap and breakpoint body command sequences
T	Echo commands read from the terminal

For example, the command

```
SET,OUTPUT,E,W,I
```

causes output types E, W, and I to be displayed at the terminal while it suppresses types D, R, and B.

The default output types are E, W, D, and I; if you do not enter a SET,OUTPUT command, these output types are displayed at the terminal. You do not need to specify type T in a SET,OUTPUT command because all user input is displayed at the terminal when it is entered.

The only output types not automatically displayed are commands executed in command sequences (types R and B). Command sequences are described in section 5. To display this output in addition to the default types, enter the command:

```
SET,OUTPUT,E,W,I,D,R,B
```

If you specify the R option on the SET,OUTPUT command, then whenever a READ command is executed, each command in the specified group or file is displayed at the terminal as it is executed. If you specify the B option, then whenever a breakpoint or trap is executed, each command in the body is displayed as it is executed.

The only output types that cannot be suppressed are the informative messages issued when breakpoints or traps are detected (these are included in type I). These messages are always displayed regardless of SET,OUTPUT specifications. Error messages (type E) can be suppressed only if you have provided for writing them to an auxiliary file with the SET,AUXILIARY command. If you attempt to suppress error messages but have not provided for writing them to an auxiliary file, CID issues an error message.

If you suppress warning messages by omitting W from the SET,OUTPUT command, CID executes all commands that would normally generate a warning message. No ? prompt is issued; CID takes the corrective action described in the warning message, responding as if you had entered a YES or OK response (described in section 3 under Error and Warning Processing).

To suppress all output to the terminal (except breakpoint and trap messages), you can issue either a SET,OUTPUT command with no option list or the command:

```
CLEAR,OUTPUT
```

The short form of CLEAR,OUTPUT is COUT.

Before entering either of these commands, however, you must provide for writing error messages to an auxiliary file.

After a CLEAR,OUTPUT command has been issued, you can restore output to default conditions with the command:

```
SET,OUTPUT,E,W,D,I
```

The SET,OUTPUT command can be used in conjunction with the SET,AUXILIARY command to suppress certain types of output to the terminal and to send that output type to an auxiliary file. The most common output to suppress is type D, the output produced by the execution of CID commands. This includes output produced by the PRINT, MAT PRINT, LIST,VALUES and DISPLAY commands, all of which can produce excessive output.

SET,AUXILIARY COMMAND

The SET,AUXILIARY command defines an auxiliary output file and specifies which types of CID output are to be written to that file. The SET,AUXILIARY command has the following form:

```
SET,AUXILIARY,lfn,type-list
```

where lfn is the name of the auxiliary file and type-list is a list of the output type codes as shown in table 4-1. The type codes can be separated by commas or entered without separators. The short form of SET,AUXILIARY is SAUX.

The SET,AUXILIARY command has no effect on output that is being displayed at the terminal. For example, the command

```
SET,AUXILIARY,FAUX,I,D
```

creates a file named FAUX and writes all informative and command output messages to the file. These messages are also displayed at the terminal unless you have entered the appropriate SET,OUTPUT command to suppress these output types.

You can change the option specifications for an auxiliary file by entering another SET,AUXILIARY command that specifies a file name and a new option list; you need not close the file beforehand.

Only one auxiliary file can be in use at a time. The QUIT command closes the auxiliary file currently in use. To close an auxiliary file before the end of a debug session, enter the command:

```
CLEAR,AUXILIARY
```

The short form of CLEAR,AUXILIARY is CAUX.

You can close an auxiliary file any time during a debug session.

The auxiliary file is a local file. After you terminate the debug session, you can display the auxiliary file at the terminal, send it to a printer, or store it on a permanent storage device. CLEAR,AUXILIARY does not rewind the file. After issuing a CLEAR,AUXILIARY you can issue a SET,AUXILIARY for the same file in the same or in a subsequent session, and the additional information will be written after the end-of-record.

A common use of the SET,AUXILIARY command is to preserve a copy of a debug session log. For example, the command

```
SET,AUXILIARY,OUTF,E,W,D,I,T
```

issued at the beginning of a debug session, writes the output types E, W, D, I, and T to file OUTF, thus creating a copy of the session exactly as displayed at the terminal. Note that when outputting to an auxiliary file, you must specify the T option to include user-entered commands in the file.

Example 1 in figure 4-2 illustrates a SET,OUTPUT command used in conjunction with a SET,AUXILIARY command to suppress output to the terminal and write it to an auxiliary file. This example suppresses all output produced by CID commands (type D), creates an auxiliary file called LGF to which this output is to be written, writes the values of all program variables to LGF, closes LGF, and resets output options to original conditions.

Example 2 in figure 4-2 illustrates a CLEAR,OUTPUT command used with a SET,AUXILIARY command. This example defines an auxiliary file named AUXF to receive error messages and output from CID commands, turns off output to the terminal (except for breakpoint and trap messages), writes program variables and contents to AUXF, restores terminal output to normal default conditions, and closes AUXF.

An example of a debug session using an auxiliary file is illustrated in figure 4-3. This session was produced by executing program TRIANGL (figure 3-3 in section 3) under CID control. In this example, an auxiliary file AFILE is defined; the D option causes output from CID commands to be sent to AFILE. A breakpoint is set at line 240 of subroutine AREA, and output to the terminal is suppressed. (Note, however, that the breakpoint message and the program output still appear.) On each pass through subroutine AREA, the breakpoint suspends execution, and LIST,VALUES is entered to write all variable names and values to the auxiliary file. After the third pass through AREA, normal output conditions are restored, the value of the variable A is displayed, and the session is terminated. File AFILE (figure 4-4) contains the output from the LIST,VALUES command. (Another way of doing this would be to include the SET,OUTPUT and LIST,VALUES commands in a breakpoint body. This would preclude the necessity of reentering these commands on each pass through the subroutine. Breakpoint bodies are described in section 5.)

```
Example 1:

      ? set,output,e,w,i
      ? set,auxiliary,lgf,d
      ? list,values
      ? clear,auxiliary
      ? set,output,e,w,i,d


Example 2:

      ? set,auxiliary,auxf,d,e
      ? clear,output
      ? list,values
      ? set,output,e,w,d,i
      ? clear,auxiliary
```

Figure 4-2. SET,OUTPUT and SET,AUXILIARY Commands

```
 CYBER INTERACTIVE DEBUG
? set,auxiliary,afile,d,e <---------- Establish auxiliary file AFILE and send all command output and
                                      error messages to this file.
? set,breakpoint,l.240

? clear,output <--------------------- Suppress output to terminal.

? go

 *B #1, AT L.240
? list,values

? go

THE AREA OF THE TRIANGLE IS  2
 *B #1, AT L.240                  <-- List variables and values while execution is suspended on each
? list,values                         pass through subroutine AREA.  Output is written to AFILE.

? go

THE AREA OF THE TRIANGLE IS  .55
 *B #1, AT L.240
? list,values

? go

THE AREA OF THE TRIANGLE IS  37.455
 *B #1, AT L.240
? set,output,e,w,d,i <--------------- Restore normal output to terminal.

? print a <-------------------------- Display value of A.  This value is also written to AFILE.

  23.7
? quit
```

Figure 4-3. Debug Session Illustrating SET,AUXILIARY, SET,OUTPUT and CLEAR,OUTPUT Commands

```
1                                                    CYBER INTERACTIVE DEBUG
0
 *B #1, AT L.240
 P.TRIANGL
 A =  2,   S1 =  2,   S2 =  2,   S3 =  2.82843,   T =  3.41421,   X1 =  0      <-- First pass.
 X2 =  2,   X3 =  0,   Y1 =  0,   Y2 =  0,   Y3 =  2
 *B #1, AT L.240
 P.TRIANGL
 A =  .55,   S1 =  1.11803,   S2 =  1.0198,   S3 =  1.7,   T =  1.91892      <-- Second pass.
 X1 =  0,   X2 =  .5,   X3 = -1,   Y1 = -1,   Y2 =  2,   Y3 =  1.2
 *B #1, AT L.240
 P.TRIANGL
 A =  37.455,   S1 =  11.0027,   S2 =  11.9854,   S3 =  6.9029              <-- Third pass.
 T =  14.9455,   X1 =  .2,   X2 = -1.3,   X3 =  5.6,   Y1 = -2.9
 Y2 =  8,   Y3 =  7.8
 *B #1, AT L.240
  23.7
```

Figure 4-4. Listing of Auxiliary File AFILE

CHAINED-TO BASIC PROGRAMS

It is not possible to CHAIN to another BASIC program when CID is in control. Since the BASIC CHAIN statement is logically equivalent to a STOP statement followed by the BASIC subsystem OLD and RUN commands, the CHAIN statement causes an END trap. To use the CHAIN statement, you must quit the current debug session and begin a new session by manually issuing the appropriate system commands to compile and execute the program called in the CHAIN statement. (Under NOS, the chained-to program will already have been made the primary file.)

REFERENCING LOCATIONS OUTSIDE YOUR BASIC PROGRAM

BASIC programs sometimes consist of a BASIC main program that calls one or more FORTRAN subroutines. CID allows you to debug both the BASIC main program and the FORTRAN subroutines in the same debug session. In order to do this, you must be familiar with the concept of the home program.

HOME PROGRAM

When a BASIC main program which calls FORTRAN subroutines is executed under CID control, execution can be suspended in the main program or in any of the FORTRAN subroutines. The default home program is the program unit in control at the time of suspension.

It is important to note that if the current home program is a FORTRAN subroutine, the BASIC CID commands LET, IF, GOTO, PRINT, and MAT PRINT cannot be used because their BASIC-like syntax is not FORTRAN compatible. Instead you must use equivalent FORTRAN CID commands with FORTRAN-like syntax. For further explanation of FORTRAN CID commands, see the CYBER Interactive Debug reference manual.

The home program concept is illustrated by the debug session in figure 4-5. (Figure 4-5 also shows one way to compile a BASIC main program with a FORTRAN subroutine for use with CID.) Breakpoints are set to suspend execution in the main program after the call to SUBA, and in SUBA itself. When execution is suspended in SUBA, SUBA is the home program and the PRINT command shows 2.0 as the value of A; when execution is suspended in the main program, 1.0 is the value of A.

In some cases, you might want to reference a location in a program unit other than the home program. One way to accomplish this is to use CID commands which allow a variable name or line number to be qualified by a program unit name.

QUALIFICATION NOTATION

Qualification notation allows you to specify a variable name or line number that occurs in a program unit other than the home program. This notation has the following forms:

P.prog_var

Denotes variable var in program unit prog.

P.prog_L.n

Denotes line n in program unit prog.

The program unit name and the variable or line number are separated by an underscore character. Qualification notation is valid for all CID commands covered in this user's guide except the PRINT, MAT PRINT, GOTO, and LET commands.

Some examples of qualification notation are as follows:

P.FACT_N

Variable N in program unit FACT.

P.SUBA_L.410

Line 410 in program unit SUBA.

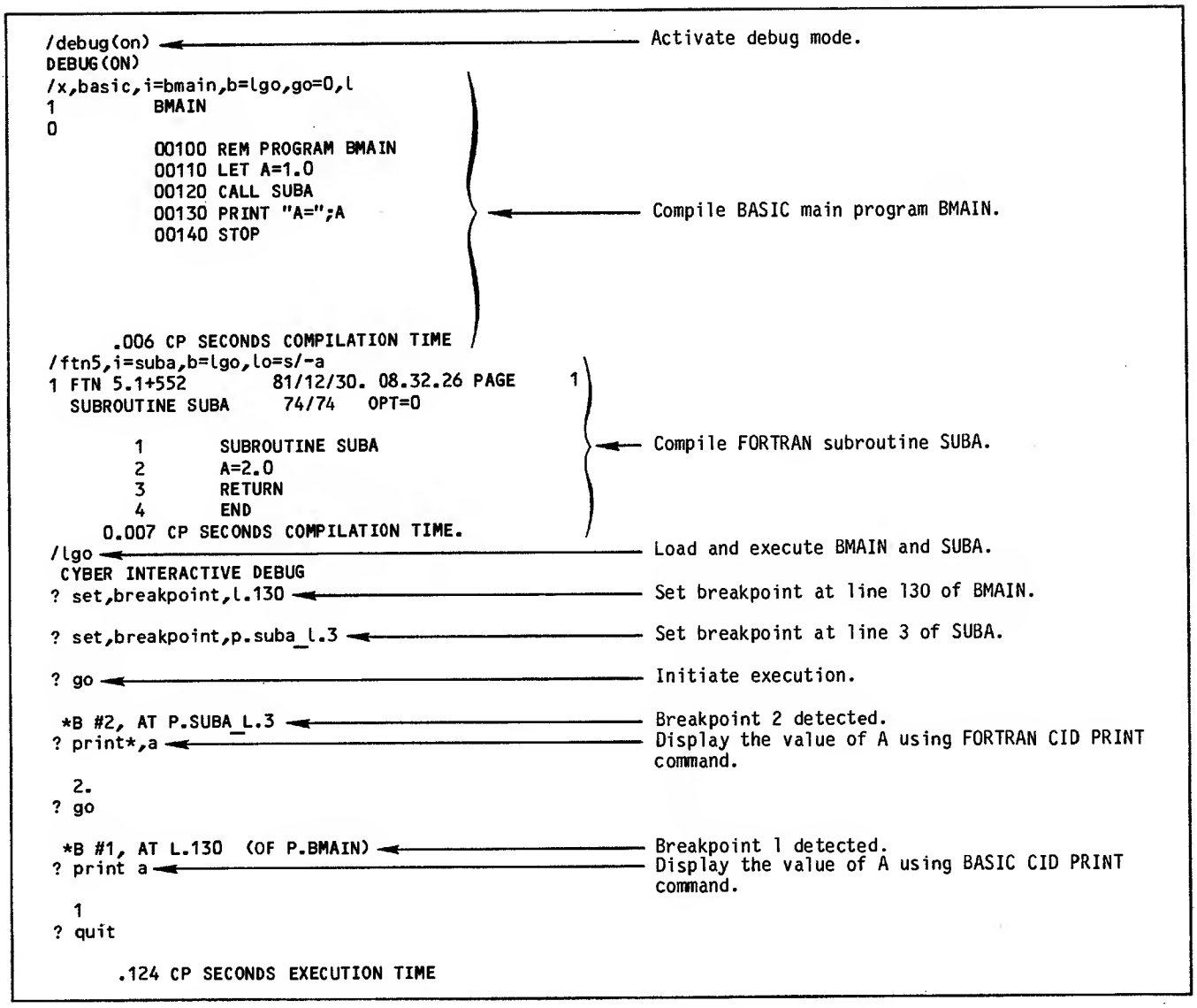

Figure 4-5. Debug Session Illustrating Home Program Concept

Qualification notation can be substituted for the normal variable or line number in CID commands for which this notation is valid, as in the following examples:

```
SET,BREAKPOINT,P.ADD_L.210
```

Sets a breakpoint at line 210 of program unit ADD.

```
SET,TRAP,STORE,P.PROC_A
```

Sets a STORE trap for variable A in program unit PROC.

Qualification notation also appears in many types of CID informative output. For example, the message

```
*B #1 AT P.NOSUB_L.140
```

indicates that a breakpoint was encountered at line 140 of the current home program unit NOSUB.

CID notation forms are summarized in table 4-2.

TABLE 4-2. CID NOTATION

Notation	Description
P.prog	Program unit prog
var	Simple or subscripted variable name
P.prog_var	Variable in program unit prog
L.n	Source line having sequence number n
P.prog_L.n	Source line having sequence number n in program unit prog

SET,HOME COMMAND

As an alternative to qualification notation or in cases where this notation is invalid, you can specify locations outside the default home program by first issuing the command

```
SET,HOME,program-unit
```

where program-unit specifies which program unit is the home program. Any unqualified variable names or line numbers specified after entering the SET,HOME command belong to program-unit. The short form of SET,HOME is SH.

It is important to note that the SET,HOME command does not alter the location where execution resumes when you issue GO; execution always resumes at the location where it was suspended, regardless of SET,HOME specification. In addition, when execution is resumed, a previous SET,HOME specification is lost, and the home program reverts to the one currently executing.

The debug session in figure 4-6, which uses the programs shown in figure 4-5, illustrates an example of the SET,HOME command. Note that on program termination, the home program is once again the main program. To print the value of A in subroutine SUBA, the SET,HOME command must be entered.

DEBUGGING AIDS FOR PROGRAMS WITH MULTIPLE PROGRAM UNITS

CID provides features that can be helpful when your BASIC main program contains a number of FORTRAN subroutine calls. The #HOME debug variable tells you the name of the program unit where execution is suspended (unless you changed the home program name with the SET,HOME command). The LIST,MAP command provides you with a concise list of subroutine names.

#HOME DEBUG VARIABLE

The #HOME debug variable is a special variable belonging to CID. This variable always contains the name of the current home program. You can display the contents of this variable with the command:

```
DISPLAY,#HOME
```

CID displays the name of the current home program in the form:

```
#HOME=P.program-unit
```

Although CID normally displays the home program name when suspension occurs in a different program unit, this variable is useful for determining the subroutine in which execution is suspended. Note, however, that if you change the home program with the SET,HOME command, #HOME contains the name of the new home program.

LIST,MAP

The LIST,MAP command, which displays load map information, is useful when your BASIC program contains many FORTRAN subroutine calls because it provides a concise list of subroutine names. This command has the forms:

```
LIST,MAP
```

Lists all modules (program units) in your field length. The list includes BASIC and FORTRAN library modules as well as user-defined modules.

```
LIST,MAP,P.name1,P.name2,...,P.namen
```

Lists the first word address (FWA), length (octal words), and all entry point names for the specified program units.

Figure 4-7 illustrates a debug session for the programs in figure 4-5 in which the LIST,MAP command is issued.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.130 <------- Set breakpoint at line 130 of home program (BMAIN).

? go <------------------------- Initiate execution.

 *B #1, AT L.130 <------------- Execution suspended at line 130.
? print a <------------------- Display value of A defined in home program (BMAIN).

  1
? set,home,suba <------------- Designate FORTRAN subroutine SUBA as home program.

? print*,a <------------------- Display value of A defined in home program (SUBA).

  2.
? go <------------------------- Resume execution (at point of suspension in BMAIN).

A= 1
 *T #17, END IN P.BMAIN L.140 <- Program terminates.
? print a <------------------- Display value of A defined in home program (BMAIN).

  1
? quit
```

Figure 4-6. Debug Session Illustrating SET,HOME Command

```
 CYBER INTERACTIVE DEBUG
? list,map
                                                                    User programs.
 DBUG.,   BMAIN,   SUBA,   BASEGEN,   BASRTS,   BASSINT,   BASSMGR
 BASOGEN,   BASERRS,   CPU.CPM,   CPU.LFM,   CPU.SYS,   UCLOAD
? list,map,p.bmain                                                  Display first word address,
                                                                    length, and entry points of
 PROGRAM - BMAIN,   FWA = 3446B,   LENGTH = 554B                    BMAIN.
 ENTRY POINTS - BASINT.,   BASSRT.,   BMAIN
? list,map,p.suba                                                   Display first word address,
                                                                    length, and entry points of
 PROGRAM - SUBA,   FWA = 4222B,   LENGTH = 21B,   ENTRY POINTS - SUBA   SUBA.
?
```

Figure 4-7. Debug Session Illustrating LIST,MAP

In many cases, you need to enter the same command or sequence of commands repeatedly during the course of a debug session. Reentering complicated assignment commands in this manner can be a time-consuming process. To eliminate this process, CYBER Interactive Debug (CID) provides you with the ability to define, save, and automatically execute sequences of commands. This feature can be used to improve debugging efficiency whenever the same group of CID commands must be entered repeatedly. Automatic command execution is commonly used when debugging loops and in multiple debug sessions that require the same commands. In addition, CID provides some special sequence commands that allow you to incorporate BASIC-like logic into command sequences. For example, sequence commands allow branching and conditional execution of CID commands.

COMMAND SEQUENCES

A command sequence, which consists of a series of CID commands, is executed automatically either when certain conditions occur or when you enter the appropriate command from the terminal.

There are three ways to establish a command sequence:

- By defining a command sequence as part of a breakpoint or trap. This causes the sequence to be executed whenever the breakpoint or trap occurs. A sequence defined in this manner is called a breakpoint body or trap body.
- By defining a command sequence called a group. A group is executed by issuing a READ command from the terminal or from another command sequence.
- By creating a file which contains a command sequence. The commands in this file are executed by issuing a READ command at the terminal or from another command sequence.

During normal execution, CID prompts you for input after a command is executed. During sequence execution, however, CID executes all the commands in the sequence without interruption. Once execution of the sequence is completed, execution of your program resumes at the point where it was suspended. You do not get control during sequence execution unless you provide for it by using the PAUSE command, which is described later in this section.

Command sequences can be nested; that is, command sequences can be called from other command sequences. However, a command sequence must have finished executing before it can be executed again. (It cannot execute itself, directly or indirectly.)

COLLECT MODE

Collect mode is a mode of execution in which CID commands are not executed immediately after they are entered, but are included in a command sequence for execution at a later time. Collect mode must be activated before you can define a breakpoint body, a trap body, or a group. The procedure for entering and leaving collect mode is described under Breakpoints and Traps With Bodies.

Commands in a sequence you are creating cannot be altered while CID is in collect mode. If you want to change a command you have entered, you must leave collect mode and proceed as described under Editing a Command Sequence, or you must reenter the entire command sequence.

MULTIPLE COMMAND ENTRY

You can enter more than one command on a single line by separating the commands by a semicolon. For example:

```
SET,BREAKPOINT,L.250;LIST,VALUES;GO
```

Note that because a semicolon can be used to separate items in the BASIC PRINT and MAT PRINT statements, the CID PRINT and MAT PRINT commands must be separated by two semicolons from the next command in a multiple command line, as in the following example:

```
SET,BREAKPOINT,L.310;MAT PRINT A;;GO
```

In interactive mode, CID does not execute the multiple command line until you press the RETURN key; it then executes the commands in the order you entered them. In collect mode, the commands are not executed immediately, but are included in the command sequence for execution at a later time.

This method of command entry is especially convenient when you are defining command sequences because you do not have to wait for an input prompt before entering each command.

SEQUENCE COMMANDS

CID provides a set of commands intended specifically for use with command sequences. These commands are summarized in table 5-1.

TABLE 5-1. SEQUENCE COMMANDS

Command	Description
EXECUTE	Resumes execution of your program
GO	Resumes the process most recently suspended
IF	Performs conditional execution of commands
JUMP	Transfers control within a command sequence to a label defined by a LABEL command
LABEL	Defines a label within a command sequence
PAUSE	Temporarily suspends execution of the current command sequence and reinstates interactive mode allowing commands to be entered from the terminal
READ	Initiates execution of a command sequence defined as a group or stored on a file; reestablishes trap, break point, and group definitions stored on a file

BREAKPOINTS AND TRAPS WITH BODIES

A body is a sequence of commands specified as part of a SET,BREAKPOINT or SET,TRAP command. To define a breakpoint or trap with a body, you must first initiate collect mode by including a left bracket as the last parameter of the SET,BREAKPOINT or SET,TRAP command. For example:

```
SET,TRAP,LINE,1.140...1.180 [
```

The bracket and the preceding parameter must not be separated by a comma; the blank separator is optional.

When the above command is entered, CID displays:

```
IN COLLECT MODE
?
```

You then enter the commands that make up the body. Each command entered while CID is in collect mode becomes part of the body. CID scans the command for syntax errors but does not execute the command. Any number of commands can be included in a body, but command sequences should be kept short and simple so that debugging the sequence does not require more time than debugging your program.

To leave collect mode and return to interactive mode, enter a right bracket in response to the ? prompt or at the end of a command line. CID then displays

```
END COLLECT MODE
?
```

and you can continue the session.

An example of defining a breakpoint with a body is shown in figure 5-1. Note that the command sequence shown in figure 5-1 can also be entered as follows:

```
SET,BREAKPOINT,L.180[LET Y=X/2.0;PRINT X,Y]
```

```
? set,breakpoint,l.180 [

IN COLLECT MODE
? let y=x/2.0
? print x,y
? ]
END COLLECT MODE
```

Figure 5-1. Breakpoint With Body

When a breakpoint or trap with a body is encountered, program execution is suspended and the commands in the body are executed automatically. Program execution then resumes at the breakpoint or trap location; CID does not give control to you upon completion of the command sequence.

When a breakpoint or trap with a body is encountered during execution, the normal breakpoint or trap message is not displayed. However, you can provide your own notification of the execution of a breakpoint or trap body by including a PRINT command in the sequence.

You do not get control during execution of a sequence unless you have provided for it by including a PAUSE command (described under Receiving Control During Sequence Execution) in the body. When the body has been executed, execution of your program automatically resumes at the location where it was suspended.

An example of the procedure for establishing a breakpoint body is illustrated in figure 5-2. The program used in this example is shown in figure 3-3 (see section 3). A breakpoint with a body is established at the RETURN statement in subroutine AREA. The breakpoint body contains the following commands:

A PRINT command to display a message at the beginning of the sequence

A DISPLAY command to display the contents of the #LINE variable which contains the current line number

Two PRINT commands to display the input values and the value of A

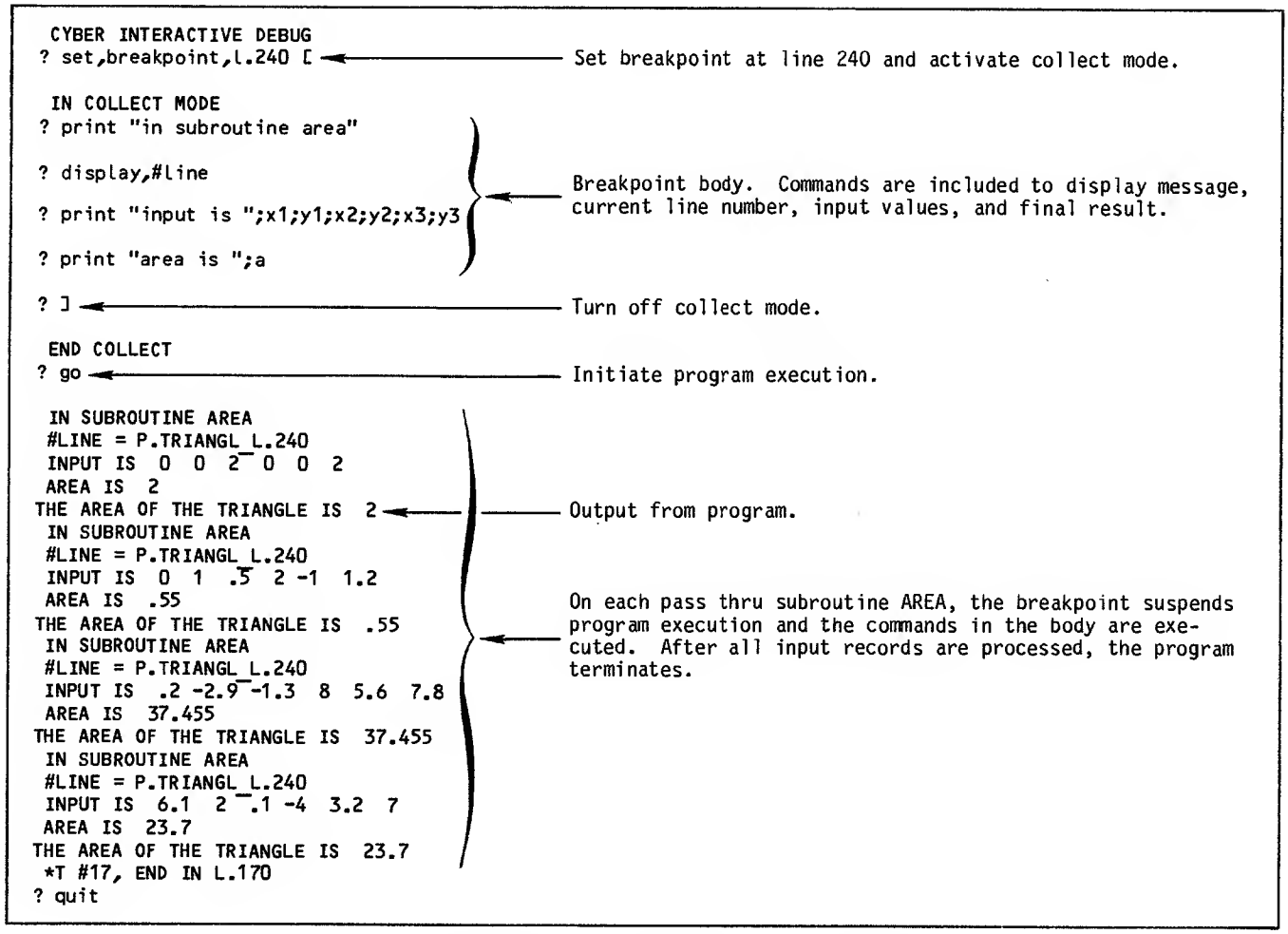

Figure 5-2. Debug Session Illustrating Breakpoint With Body

Subroutine AREA is called four times; each time the breakpoint is detected, the commands in the sequence are executed.

Figure 5-3 shows how a trap body can be used to trace program execution. In this session, a LINE trap with a body is set. The trap body displays the value of the debug variable #LINE. Figure 5-3 uses the program shown in figure 3-13 (see section 3).

DISPLAYING BREAKPOINTS AND TRAPS WITH BODIES

You can display a list of the commands in a breakpoint body by entering one of the following forms of the LIST,BREAKPOINT command:

LIST,BREAKPOINT,L.m,...,L.n

Displays the complete definitions, including the bodies (if any), of the breakpoints at the specified lines, m,...,n.

LIST,BREAKPOINT,#n1,#n2,...,#nm

Displays the complete definitions, including the bodies (if any), of the breakpoints having numbers n1, n2,...,nm; breakpoint numbers are assigned by CID when the breakpoints are established.

Other forms of the LIST,BREAKPOINT command list the breakpoint location but not the commands in the body.

Following are some examples of the LIST,BREAKPOINT command.

LIST,BREAKPOINT,#1,#5,#6

Lists the locations and bodies of breakpoints 1, 5, and 6.

LB,L.190

Lists the location and body of the breakpoint at line 190.

Figure 5-4 illustrates a LIST,BREAKPOINT command for the breakpoint established in figure 5-2.

To display a list of the commands in a trap body, enter the command:

LIST,TRAP,#n1,#n2,...,#nm

This form of the LIST,TRAP command displays the trap types, locations, and bodies (if any) of the traps having numbers n1, n2,...,nm; the trap numbers are assigned by CID when the traps are established. Other forms of the LIST,TRAP command list only the trap type and location.

```
 CYBER INTERACTIVE DEBUG
? set,trap,line,* [display,#line]  <-- Set LINE trap with body.

? go  <-- Initiate execution.

 #LINE = P.ARRYB_L.100
 #LINE = P.ARRYB_L.110
 #LINE = P.ARRYB_L.120
 #LINE = P.ARRYB_L.130
 #LINE = P.ARRYB_L.140
 #LINE = P.ARRYB_L.150
 #LINE = P.ARRYB_L.200
 #LINE = P.ARRYB_L.210
 #LINE = P.ARRYB_L.220
 #LINE = P.ARRYB_L.230
 #LINE = P.ARRYB_L.220
 #LINE = P.ARRYB_L.230
 #LINE = P.ARRYB_L.220
 #LINE = P.ARRYB_L.230
 #LINE = P.ARRYB_L.220
 #LINE = P.ARRYB_L.230
 #LINE = P.ARRYB_L.220
 #LINE = P.ARRYB_L.230  <-- Output from trap body shows program flow.
 #LINE = P.ARRYB_L.240
 #LINE = P.ARRYB_L.160
 #LINE = P.ARRYB_L.170
 #LINE = P.ARRYB_L.200
 #LINE = P.ARRYB_L.250
 #LINE = P.ARRYB_L.260
 #LINE = P.ARRYB_L.270
 #LINE = P.ARRYB_L.260
 #LINE = P.ARRYB_L.270
 #LINE = P.ARRYB_L.260
 #LINE = P.ARRYB_L.270
 #LINE = P.ARRYB_L.260
 #LINE = P.ARRYB_L.270
 #LINE = P.ARRYB_L.260
 #LINE = P.ARRYB_L.270
 #LINE = P.ARRYB_L.280
 #LINE = P.ARRYB_L.180
 *T #17, END IN L.180
? quit
```

Figure 5-3. Debug Session Illustrating Trap With Body

```
? list,breakpoint,#1  <-- Display the complete definition of breakpoint #1 including the
                          breakpoint location and all commands in the breakpoint body.
 *B #1 = L.240
 SET,BREAKPOINT,L.240 [
 PRINT "IN SUBROUTINE AREA"
 DISPLAY,#LINE
 PRINT "INPUT IS ";X1;Y1;X2;Y2;X3;Y3
 PRINT "AREA IS ";A
 ]
?
```

Figure 5-4. Debug Session Illustrating LIST,BREAKPOINT Command for Breakpoint With Body

For example,

```
LIST,TRAP,#2,#5
```

lists the type, location, and body of the traps numbered 2 and 5.

GROUPS

A group is a sequence of commands established and assigned a name during a debug session, but not explicitly associated with a breakpoint or trap. A group exists until you clear it or terminate the debug session and is executed by entering an appropriate READ command. The command to establish a group is as follows:

```
SET,GROUP,name [
```

where name is the name by which you reference the group. The left bracket activates collect mode, as with breakpoint and trap bodies. All CID commands subsequently entered become part of the sequence

until you terminate the sequence by entering a right bracket. The short form of SET,GROUP is SG.

The command to execute a group is as follows:

```
READ,name
```

where name is the group name assigned in the SET,GROUP command. You can issue a READ command directly from the terminal or from another command sequence. In response to a READ command, CID executes the commands in the group. After a group has been executed, control returns to CID (if the READ was entered from the terminal) or to the next command in the sequence that issued the READ.

A group can be used when the same sequence of commands is to be executed at different locations in a program because a group, unlike a breakpoint or trap body, can be executed at any time during the debug session. Figure 5-5 shows an example of a simple group definition.

```
? set,group,grpa [

 IN COLLECT MODE
? let x=y+z
? print x,y,z
? ]
 END COLLECT
?
```

Figure 5-5. SET,GROUP Command Example

The command sequence in figure 5-5 is executed by entering the command:

```
READ,GRPA
```

When a group is established, it is assigned a number in the same manner as breakpoints and traps. A group can be referred to by number or name in the LIST, CLEAR, and SAVE commands.

You can list the names and numbers of the groups currently defined by entering one of the following forms of the LIST,GROUP command:

```
LIST,GROUP,*
```

Lists the names and numbers of all groups defined for the current debug session. This statement does not list the commands contained in the groups.

```
LIST,GROUP,name-list
```

Lists the commands contained in the groups specified in name-list. Group names are separated by commas.

```
LIST,GROUP,#n1,#n2,...,#nm
```

Lists the commands contained in the groups identified by the specified numbers.

The short form of LIST,GROUP is LG.

Normally, a group exists for the duration of a debug session. You can remove existing groups from the current debug session by entering one of the following forms of the CLEAR,GROUP command:

```
CLEAR,GROUP,*
```

Removes all currently defined groups.

```
CLEAR,GROUP,name-list
```

Removes the specified groups.

```
CLEAR,GROUP,#n1,#n2,...,#nm
```

Removes the groups identified by the specified numbers.

The short form of CLEAR,GROUP is CG.

Figures 5-6 and 5-7, which use the program shown in figure 3-13 (see section 3), illustrate debug sessions using groups. In figure 5-6, two breakpoints are set in subroutine SETB. When either breakpoint is reached, the READ command is issued from the terminal. In figure 5-7, the same breakpoints are established except that a body containing a READ command is defined for each breakpoint. This causes the body to be executed automatically with no intervention from the user when the breakpoints are encountered. By defining a single group instead of defining a body for each breakpoint, you need enter the command sequence only once.

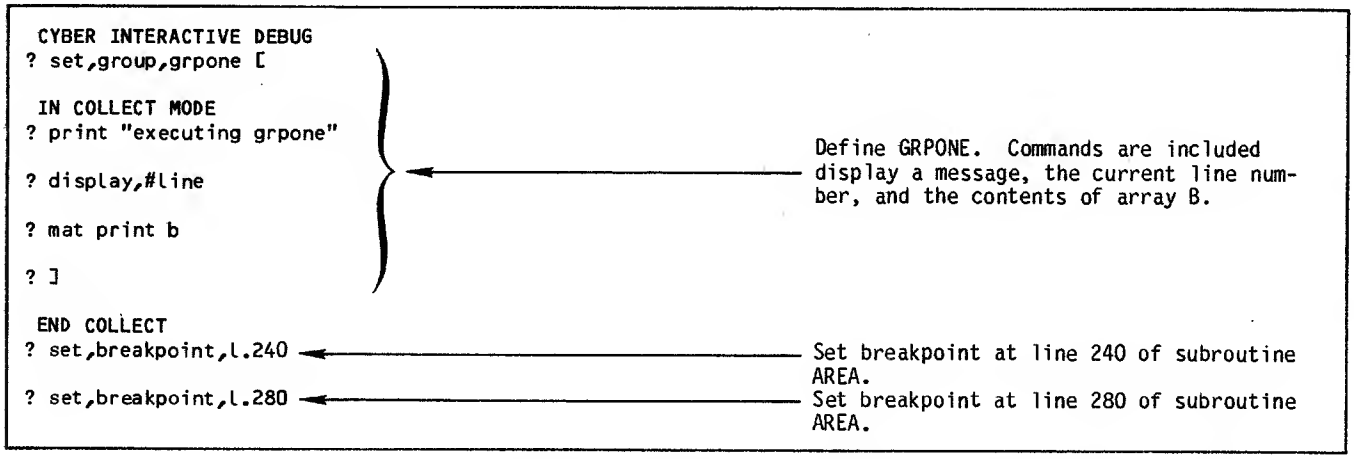

Figure 5-6. Debug Session Illustrating Group Execution Initiated at Terminal (Sheet 1 of 2)

```
? list,breakpoint,#1,#2 <-------------------- List breakpoint definitions.

 *B #1 = L.240,   *B #2 = L.280
? go <-------------------- Initiate execution.

 *B #1, AT L.240 <-------------------- Breakpoint suspends execution at line 240.
? read,grpone <-------------------- Execute the commands in GRPONE.

 EXECUTING GRPONE
 #LINE = P.ARRYB_L.240
  1            1            1            1            1



? go <-------------------- Resume execution.

 *B #2, AT L.280
? read,grpone <-------------------- Execute the commands in GRPONE.

 EXECUTING GRPONE
 #LINE = P.ARRYB_L.280
 -1           -1           -1           -1           -1



? quit
```

Figure 5-6. Debug Session Illustrating Group Execution Initiated at Terminal (Sheet 2 of 2)

```
 CYBER INTERACTIVE DEBUG
? set,group,grpone [
 IN COLLECT MODE
? print "executing grpone"
? display,#line                    <-- Define GRPONE. Commands are included
? mat print b                          to display a message, the current line
? ]                                    number, and the contents of array B.

 END COLLECT
? set,breakpoint,l.240 [read,grpone]   <-- Set breakpoints at lines 240 and 280.
? set,breakpoint,l.280 [read,grpone]       Define a body for each breakpoint which
                                           contains a READ command to initiate exe-
                                           cution of the commands in GRPONE.
? list,breakpoint,#1,#2 <-------------------- List the definitions of the breakpoints.

 *B #1 = L.240
 SET,BREAKPOINT,L.240 [
 READ,GRPONE]
 *B #2 = L.280
 SET,BREAKPOINT,L.280 [
 READ,GRPONE]
? go <-------------------- Initiate program execution.

 EXECUTING GRPONE
 #LINE = P.ARRYB_L.240
  1            1            1            1            1
                                                          <-- The breakpoints of lines 240 and 280
                                                              suspend program execution and the READ
EXECUTING CRPONE                                              commands in the bodies are automatically
#LINE = P.ARRYB_L.280                                         executed, causing the commands in GRPONE
 -1           -1           -1           -1           -1       to be executed.


 *T #17, END IN L.180
? quit
```

Figure 5-7. Debug Session Illustrating Group Execution Initiated From Breakpoint Body

In figure 5-7, note that there are three levels of execution: the program, the breakpoint body, and the group. When the breakpoint is reached, the program is suspended and execution of the breakpoint body is initiated. When the READ command is encountered, execution of the breakpoint body is suspended while the group is executed. When execution of the group is complete, execution of the suspended breakpoint body resumes at the command following the READ. When execution of the breakpoint body is complete, execution of the suspended program resumes.

Groups are especially useful when the same sequence of commands is to be executed at more than one location within a program. An example of this is illustrated in figures 5-8 and 5-9. The program MATOP defines two matrices and calls the BASIC arithmetic matrix operations to add, subtract, and multiply the matrices and store the results in an array called M3. The purpose of the debug session (figure 5-9) is to print the contents of M3 after each matrix operation is performed. To accomplish this, a group named MTRX is defined to contain the appropriate MAT PRINT command. After each subroutine call, a breakpoint with a body containing a command to execute the commands in group MTRX is set. When each breakpoint is encountered, the group commands are automatically read and executed.

```
00100 REM PROGRAM MATOP
00110 OPTION BASE 1
00120 DIM M3(3,3)
00130 MAT READ M1(3,3)
00140 MAT READ M2(3,3)
00150 MAT M3=M1+M2
00160 MAT M3=M1-M2
00170 MAT M3=M1*M2
00180 DATA 1,2,3,4,5,6,7,8,9
00190 DATA 10,11,12,13,14,15,16,17,18
00200 END
```

Figure 5-8. Program MATOP

```
 CYBER INTERACTIVE DEBUG
? set,group,mtrx [

 IN COLLECT MODE
? print "printing the contents of m3"

? display,#line

? mat print m3

? ]

 END COLLECT
? set,breakpoint,l.160 [read,mtrx]

? set,breakpoint,l.170 [read,mtrx]

? set,breakpoint,l.180 [read,mtrx]

? list,group,mtrx

 *G #1 = MTRX
 SET,GROUP,MTRX [
 PRINT "PRINTING THE CONTENTS OF M3"
 DISPLAY,#LINE
 MAT PRINT M3
 ]
? go

 PRINTING THE CONTENTS OF M3
 #LINE = P.MATOP_L.160
  11            13            15

  17            19            21

  23            25            27
```

Define group MTRX. Command is included to display the values of array M3.

Set breakpoints at lines 160, 170, and 180. In each breakpoint body, include a READ command to initiate execution in group MTRX.

Display the definition of group MTRX.

Initiate program execution.

Breakpoint suspends execution at line 160. READ command is executed, and control returns in program.

Figure 5-9. First Debug Session for Program MATOP (Sheet 1 of 2)

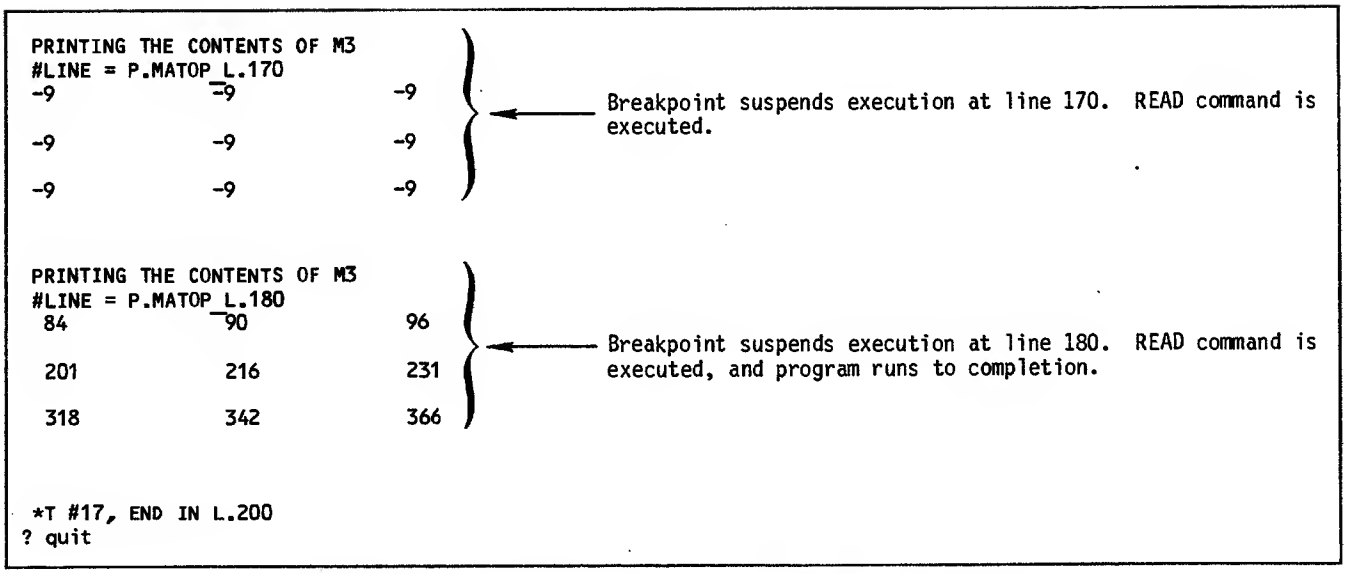

Figure 5-9. First Debug Session for Program MATOP (Sheet 2 of 2)

The debug session in figure 5-10 is identical to the debug session in figure 5-9 except that in figure 5-10 the command READ,MTRX is issued from the terminal instead of a breakpoint body. Note that when the READ command is executed in the breakpoint body (figure 5-9), program execution continues after the group commands are executed. However, when the READ command is entered at the terminal (figure 5-10), control returns to CID after the group commands are executed, and program execution must be resumed with a GO command.

ERROR PROCESSING DURING SEQUENCE EXECUTION

When CID is in collect mode and you are defining a command sequence, CID scans each command you enter for syntax errors. If a syntax error is detected, CID displays an error message followed by a ? prompt. You can then reenter the command. However, other errors such as nonexistent line number or variable name cannot be detected until CID attempts to execute the command.

CID issues normal error and warning messages during sequence execution. When an error or warning condition is detected, CID suspends execution of the sequence and issues a message followed by an input prompt (? for error messages; OK? for warning messages) on the next line. You can instruct CID to disregard the command, replace the command with another command, or, in the case of warning messages, execute the command. The ways in which you can respond to error and warning messages are summarized as follows:

OK or Yes

CID executes the command (for warning messages only).

NO

CID disregards the command. Execution resumes at the next command in the sequence.

NO,SEQ

CID disregards the command and all remaining commands in the sequence.

Any other CID command

CID executes the specified command line in place of the current command, and resumes execution of the sequence.

An example of error processing during sequence execution is illustrated in figure 5-11. During execution of group CGRP, CID issues three error messages. After each message is issued, CID gives you control. In response to the first error message, a new command, which will be executed in place of the incorrect command, is entered. In response to the second error message, NO is entered, instructing CID to ignore the incorrect command and resume execution of the sequence. In response to the third error message, NO,SEQ is entered, instructing CID to disregard the incorrect command and all remaining commands in the sequence and to give you control.

```
 CYBER INTERACTIVE DEBUG
? set,group,mtrx [

 IN COLLECT MODE
? print "printing the contents of m3"

? display,#line

? mat print m3

? ]

 END COLLECT
? set,breakpoint,l.160

? set,breakpoint,l.170

? set,breakpoint,l.180

? go

 *B #1, AT L.160
? read,mtrx

 PRINTING THE CONTENTS OF M3
 #LINE = P.MATOP_L.160
  11              13              15

  17              19              21

  23              25              27


? go

 *B #2, AT L.170
? read,mtrx

 PRINTING THE CONTENTS OF M3
 #LINE = P.MATOP_L.170
 -9              -9              -9

 -9              -9              -9

 -9              -9              -9


? go

 *B #3, AT L.180
? read,mtrx

 PRINTING THE CONTENTS OF M3
 #LINE = P.MATOP_L.180
  84              90              96

  201             216             231

  318             342             366


? quit
```

← Define a group to print array M3.

← Set breakpoints at lines 160, 170 and 180.

← Initiate execution of group while execution is suspended at line 160.

← Resume program execution.

← Initiate execution of group while execution is suspended at line 170.

← Resume program execution.

← Initiate execution of group while execution is suspended at line 180.

Figure 5-10. Second Debug Session for Program MATOP

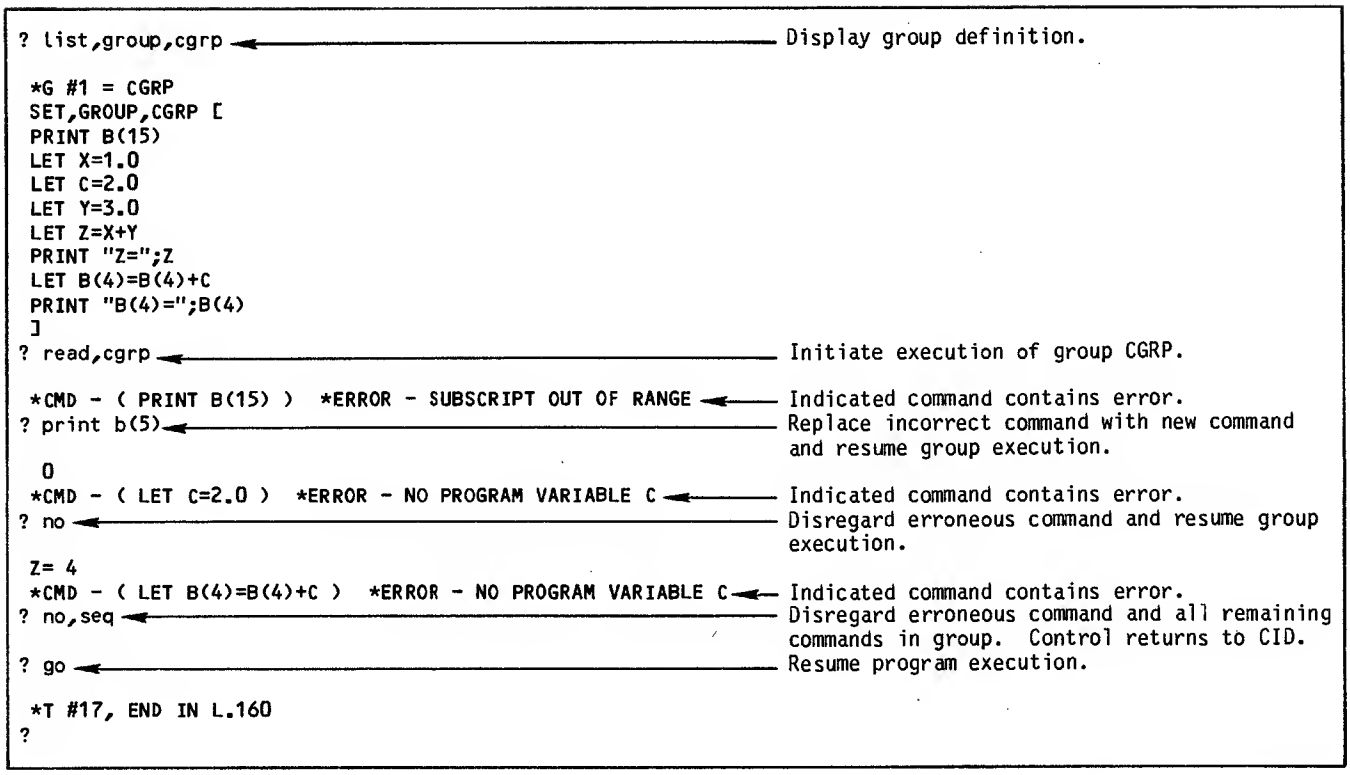

Figure 5-11. Debug Session Illustrating Error Processing During Sequence Execution

RECEIVING CONTROL DURING SEQUENCE EXECUTION

Normally, a command sequence executes to completion without returning control to CID. In some instances, however, you could want to temporarily gain control during execution of a sequence for the purpose of entering other commands and then resume execution. You can use the PAUSE command to suspend execution of a sequence and the GOTO (described in section 3), GO, or EXECUTE commands to resume execution.

PAUSE COMMAND

The purpose of the PAUSE command is to suspend execution of a command sequence. The formats of the PAUSE command are as follows:

```
    PAUSE
and
    PAUSE,"string"
```

where string is any string of characters. When CID encounters this command in a sequence, execution of the sequence is suspended and CID gets control. If a string is specified, the character string is displayed when the PAUSE command is executed.

The PAUSE command is meaningful only in a command sequence; if entered directly from the terminal, it is ignored.

When a PAUSE command is encountered in a breakpoint or trap body, CID displays the breakpoint or trap message followed by any string of characters included in the PAUSE command. CID then prompts you for input; you can enter any valid CID command.

GO AND EXECUTE COMMANDS

The functions of the GO and EXECUTE commands are identical except when issued following suspension of a command sequence. In this case, the functions of the GO and EXECUTE commands are as follows:

GO resumes execution of the suspended sequence.

EXECUTE causes an immediate exit from the sequence and resumes execution of the program.

The debug session in figure 5-12 illustrates the PAUSE, GO, and EXECUTE commands. This session was produced by executing program TRIANGL, shown in figure 3-3 (see section 3), under CID control. The purpose of this session is to suspend execution at the beginning of the subroutine AREA in order to display the input values and change them if necessary, and to suspend execution at the end of the subroutine in order to display the calculated area.

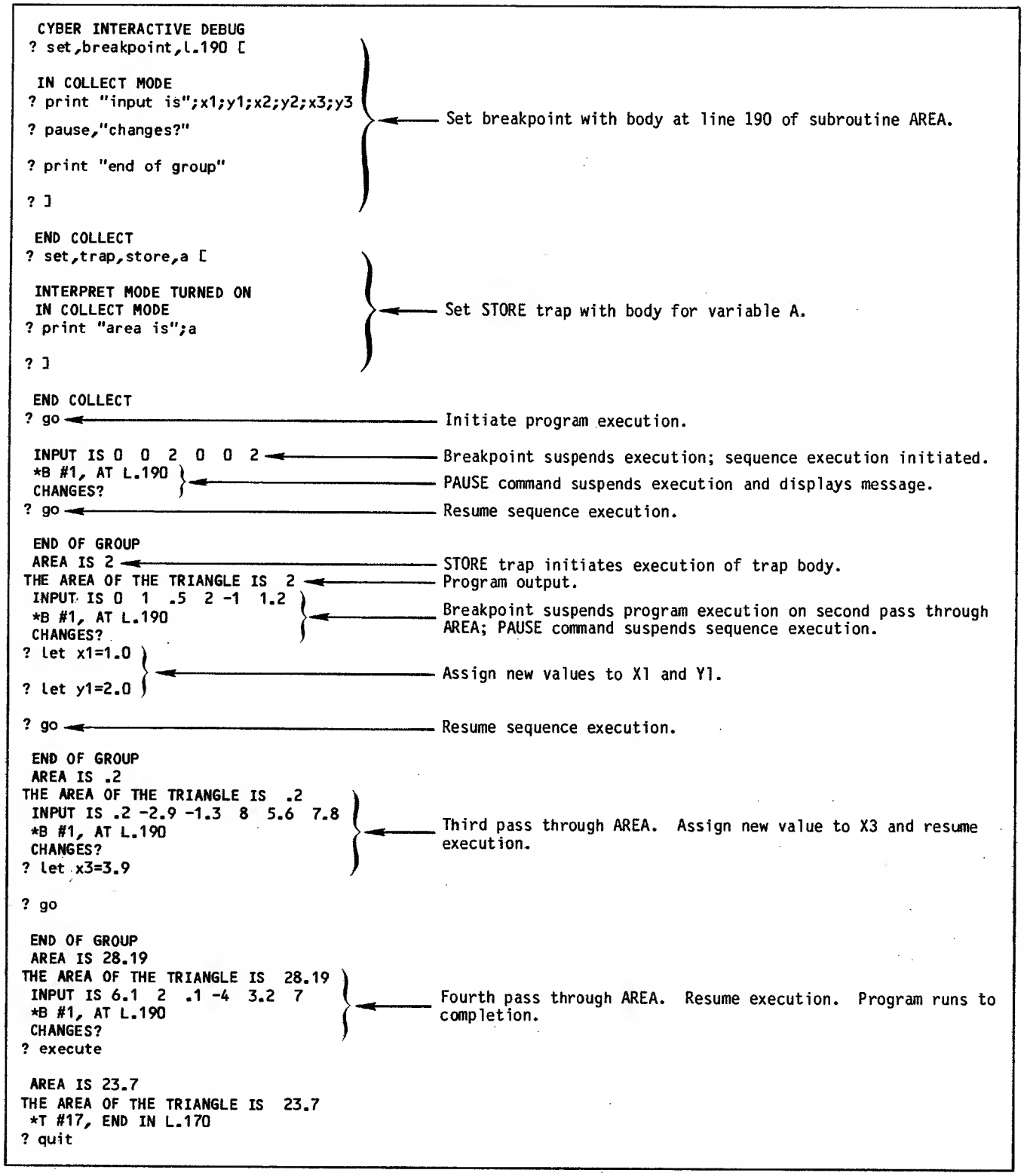

Figure 5-12. Debug Session Illustrating PAUSE, GO, and EXECUTE Commands

To accomplish this, a breakpoint with a body is set at line 190 in subroutine AREA. Three commands are included in the body: two PRINT commands and a PAUSE command. A STORE trap is established for the variable A. A body containing a command to print the value of A is defined for this trap. On each of the four passes through subroutine AREA, the commands in the sequence are executed automatically. When the PAUSE command is detected on the first pass, GO is entered to resume sequence execution. On the next two passes through the subroutine, while sequence execution is suspended because of the PAUSE command, LET commands are entered to change the values of some of the input variables. When the PAUSE command is detected on the fourth pass, EXECUTE is entered to resume program execution immediately. Note that this time the second PRINT command in the breakpoint body is not executed.

CONDITIONAL EXECUTION OF CID COMMANDS

CID allows conditional execution of commands in much the same manner as BASIC allows for conditional execution of statements. CID provides an IF command that is similar to the BASIC IF statement and a JUMP command which must be used with a LABEL command.

IF COMMAND

The IF command is used to control the selection of CID commands based on a comparison of program variables or computed variables.

The format of the IF command is as follows:

```
IF relexp THEN command
```

where relexp is a relational expression and command is any valid CID command. If the relational expression is true, the command is executed.

The form of a relational expression is the same as the form in BASIC. The following relational operators are valid: =, >, <, <>, <=, and >=.

The logical operators AND, OR and NOT can be used to connect simple relational expressions. IF THEN ELSE is not supported.

The following restrictions apply to the use of the IF command:

- Only variables defined in the current home program can appear.
- CID debug variables are not allowed.
- Function references and exponentiation are not allowed.

Although the consequent command in an IF command can be any valid CID command, it is usually a PRINT, LET, or GOTO command, as in the following examples:

```
IF X>Y+Z THEN PRINT "VALUES ARE ";X,Y
```

If X is greater than Y+Z, print the values of X and Y.

```
IF I=1 THEN LET Z=X*2.0
```

If I is equal to 1, the value X times 2.0 replaces the current value of Z.

```
IF A>0.0 AND B<0.0 THEN GOTO 150
```

If the value of A is greater than zero and the value of B is less than zero, control transfers to line 150 of the program.

Although you can issue an IF command from the terminal, this command is especially useful in command sequences. You can use the IF command together with the GO, GOTO, or EXECUTE commands to perform a test and conditionally transfer control to another command in the sequence, or to exit from the sequence as in the following examples:

```
IF A>B OR A<C THEN GO
```

If the value of A is greater than the value of B or less than the value of C, exit from the current sequence and resume execution of the most recently suspended process.

```
IF I<>0.0 THEN GOTO 270
```

If the value of I is not equal to zero, exit from the current sequence and resume program execution at statement 270.

```
IF X+T<Y+S THEN EXECUTE
```

If the value of X+T is less than the value of Y+S, exit from the current sequence and resume program execution.

JUMP AND LABEL COMMANDS

The JUMP and LABEL commands are used to transfer control within a sequence. The format of the JUMP command is as follows:

```
JUMP,name
```

where name is a label declared in a LABEL command. When CID encounters a JUMP command, control transfers to the command following the label.

A label is established within a command sequence by the command

```
LABEL,name
```

where name is a string of one through seven letters or digits. The LABEL command is not executed by CID; its sole purpose is to provide a destination for a JUMP command. When a JUMP command is executed, control transfers to the command following the LABEL command.

The JUMP command can be used in conjunction with the IF command to perform a conditional branch, as in the command sequence example shown in figure 5-13. In this example, if the value of X is less than 100.0, 1.0 is added to X and program execution resumes; if X is not less than 100.0, X is set to 0.0 and program execution resumes.

A debug session using the IF, JUMP, and LABEL commands is shown in figure 5-14. The program executed to produce this session appears in section 3 (figure 3-13). The purpose of this session is to suspend program execution at the beginning of subroutine SETB and store the value 3.0 into each word of the array B if K is equal to 3; otherwise, execution is to proceed normally. To accomplish this, a breakpoint with a body is set at line 200 in subroutine SETB. The first command in the body tests K. If K is not equal to 3, program execution resumes at line 200; otherwise, execution of the sequence continues. The remaining commands of the sequence constitute a loop that stores 3.0 into each word of B. The variable K is used as an index and counter because it is not required by the program. When K is greater than the array dimension N, program execution resumes at line 280. A breakpoint is set at line 150 in the program (the first GOSUB statement) so that K can be assigned a value of 3.

At this point, you are probably aware that command sequences using the conditional execution capability can become quite complicated. You should, however, attempt to keep sequences short and simple so that debugging the sequence does not require more time than debugging your program.

```
 IN COLLECT MODE
? if x<100.0 then jump,lab1
? let x=0.0
? go
? label,lab1
? let x=x+1.0
? ]
 END COLLECT
```

Figure 5-13. Example of JUMP and LABEL Commands

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.200 [ <---- Set breakpoint at line 200 and enter collect mode.

 IN COLLECT MODE
? if k<>3 then go <---------- If K is not equal to 3, resume program execution.

? let k=1

? label,lbla <--------------- Define label LBLA.

? let b(k)=3.0

? print "b(";k;")=";b(k)

? let k=k+1
                              } <---- Increment counter. If K is greater than N, then all elements of array B
? if k>n then goto 280        }       have been set to 3; therefore, resume program execution. Otherwise, make
                              }       another pass through sequence.
? jump,lbla                   }

? ]

 END COLLECT
? set,breakpoint,l.150
                              } <---- Suspend execution at line 150 and assign new value to K.
? go                          }
                              }
 *B #2, AT L.150              }
? let k=3

? go

 B( 1 )= 3
 B( 2 )= 3
 B( 3 )= 3 <---------------- Breakpoint suspends execution at line 200, sequence commands execute.
 B( 4 )= 3
 B( 5 )= 3
 *T #17, END IN L.180 <------ Program runs to completion.
? quit
```

Figure 5-14. Debug Session Illustrating JUMP and LABEL Commands

COMMAND FILES

In addition to executing command sequences established within a debug session, you can execute command sequences stored on a separate file. You can create such a file using a text editor and include any sequence of CID commands in the file. Command files can also be created with the SAVE command (discussed under Saving Breakpoint, Trap, and Croup Definitions). There are two reasons why you might want to create a separate file of CID commands:

By storing commands on a file, you have a permanent copy of the command sequence that can be used for future debug sessions.

Editing a file of commands using a text editor is easier than editing a sequence of commands in a group or body while executing under CID control. (See Editing a Command Sequence.)

To execute the commands in a file, enter the command

```
READ,lfn
```

where lfn is the file name. CID reads the file and automatically executes the commands in the same manner as it does for a group. When execution of the commands is complete, program execution remains suspended and control returns to you. To resume program execution, enter CO.

If a command sequence is to be executed many times in a single session, a more efficient method of executing the commands is to create a command file containing a SET,GROUP command and to include the command sequence in the group. When the file is read by the READ command, the SET,GROUP command is automatically executed and the command sequence is established as a group within the debug session. The group can subsequently be executed without reading the file again.

For example, if, in figure 5-15 example 1, the file containing the listed commands is created by means of a text editor and assigned the name COMF, the command READ,COMF must be issued whenever the sequence is to be executed. If, instead, the file in example 2 is created, the command READ,COMF reads the file and causes the SET,GROUP command to be executed, establishing GRPX for the current session. Thereafter, the command READ,GRPX executes the commands in the group and the file COMF is read only once.

Example 1:

```
LET X1=Y1+Z1
LET X2=Y2+Z2
PRINT X1,X2
```

Example 2:

```
SET,GROUP,GRPX [
LET X1=Y1+Z1
LET X2=Y2+Z2
PRINT X1,X2
]
```

Figure 5-15. SET,GROUP Example

The use of text editors under NOS and NOS/BE to create and edit files containing CID commands is described under Editing a Command Sequence.

SAVING BREAKPOINT, TRAP, AND GROUP DEFINITIONS

As with other CID commands, command sequences exist only for the duration of the session in which they sre defined. CID provides the capability of saving breakpoint, trap, and group definitions on a separate file. You can print this file or make it permanent. There are two reasons for copying CID definitions to a file:

To preserve a copy of the definitions for use in the current or subsequent debug sessions

To make it easier to edit command sequences with the system text editor

The commands to save CID definitions are as follows:

SAVE,BREAKPOINT,lfn,list

Copies to file lfn the definitions of the breakpoints specified in list; list is an optional list of breakpoint locations (L.n) or breakpoint numbers (#n) separated by commas. If you specify an asterisk or omit list, all breakpoints are saved. The short form of SAVE,BREAKPOINT is SAVEB.

SAVE,TRAP,lfn,type,scope

Copies to file lfn the trap definitions of the specified type defined for the specified scope. Type and scope are optional and are the same as for the SET,TRAP command. If you specify an asterisk or omit type and scope, all existing traps are saved. The short form of SAVE,TRAP is SAVET.

SAVE,CROUP,lfn,list

Copies to file lfn the groups specified in list; list is an optional list of group names or numbers (#n) separated by commas. If you specify an asterisk or omit list, all groups defined for the current session are saved. The short form of SAVE,CROUP is SAVEG.

The SAVE command copies the complete definition of the specified breakpoints, traps, or groups to the specified file. (The definition of a breakpoint, trap, or group includes the SET command and any other commands in the body.)

You can combine breakpoint, trap, and group definitions on a single file by specifying the same file name for multiple SAVE commands. A single READ command reestablishes all the definitions stored in the file. Another way to combine definitions on a single file is to enter the command:

```
SAVE,*,lfn
```

This command copies all existing breakpoint, trap, and group definitions to the specified file.

Some examples of SAVE commands are as follows:

```
SAVE,BREAKPOINT,SBPF,*
```

Copies to file SBPF all existing breakpoints.

```
SAVE,BREAKPOINT,BPFILE,L.140,L.320
```

Copies to BPFILE the definitions of the breakpoints established at lines 140 and 320 of the program.

```
SAVEB,FILEA,#2,#5
```

Copies to FILEA the definitions of breakpoints #2 and #5.

```
SAVE,TRAP,TFILE,*
```

Copies to TFILE all existing traps.

```
SAVET,TTT,LINE,*
```

Copies to TTT the definitions of all LINE traps established in the program.

```
SAVE,GROUP,GFIL,WRT,RDD,GRPX
```

Copies to GFIL the definitions of the groups named WRT, RDD, and GRPX.

```
SAVEG,NFILE,AREA,XYZ,NEWT
```

Copies to NFILE the definitions of the groups named AREA, XYZ, and NEWT.

The file on which the definitions are saved is a local file. If you want to access these definitions after logging out, you must make the file permanent.

Definitions stored on a file can be altered (as described under Editing a Command Sequence) and then restored in the current or a subsequent session. The command to restore the definitions stored on a file is as follows:

```
READ,lfn
```

where lfn is the file containing the definitions. You can issue a READ command in the current session or in a later session. If a READ,lfn is issued in the current session and the definitions previously saved on lfn have not been removed by the appropriate CLEAR command, CID displays a message of the form:

```
*WARN - EXISTING BREAKPOINTS WILL BE REDEFINED
OK?
```

A positive response (YES or OK) causes the existing definitions to be redefined according to the information in the file; a negative response (NO) causes the read command to be ignored.

Note that the READ command restores only the definitions stored in the specified file; it does not cause the commands in the definitions to be executed.

The following READ commands assume that GFIL and TTT are as defined in the preceding examples:

```
READ,TTT
```

Restores the LINE trap definition contained in file TTT.

```
READ,GFIL
```

Restores the group definitions contained in file GFIL.

The two debug sessions shown in figure 5-16 illustrate the SAVE command. The program shown in figure 3-3 (see section 3) is executed to produce these sessions. Two breakpoints with bodies are set and then execution is initiated. A breakpoint, with no body, is established to suspend execution before the next record is read. The program reads the first record stored in file TRARDAT, executes both command sequences, and suspends execution when the breakpoint at line 160 is detected. A SAVE,BREAKPOINT command is issued to save the current breakpoint definitions in file TRFILE. The debug session is terminated and a new session is initiated. The command READ,TRFILE restores the breakpoints for the new session. A listing of file TRFILE is shown in figure 5-17.

```
First Session:  Breakpoint Definitions Saved.

 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.140 [

 IN COLLECT MODE
? display,#line
                                  <-- Set breakpoint with body at line 140.
? list,values

? ]

 END COLLECT
? set,breakpoint,l.160  <-- Set breakpoint at line 160.
```

Figure 5-16. Debug Sessions Illustrating SAVE Command (Sheet 1 of 2)

```
? set,breakpoint,l.240 [

 IN COLLECT MODE
? print "input is";x1;y1;x2;y2;x3;y3

? print "area is ";a

? ]

 END COLLECT
? go

 #LINE = P.TRIANGL_L.140
 P.TRIANGL
 A = 0,  S1 = 0,  S2 = 0,  S3 = 0,  T = 0,  X1 = 0,  X2 = 2
 X3 = 0,  Y1 = 0,  Y2 = 0,  Y3 = 2
 INPUT IS 0  0  2  0  0  2
 AREA IS  2
THE AREA OF THE TRIANGLE IS  2
 *B #2, AT L.160
? save,breakpoint,trfile,*

? quit



Second Session:  Breakpoint Definitions Restored.

 CYBER INTERACTIVE DEBUG
? read,trfile

? list,breakpoint,*

 *B #1 = L.140 ,   *B #2 = L.160,   *B #3 = L.240
? go

 #LINE = P.TRIANGL_L.140
 P.TRIANGL
 A = 0,  S1 = 0,  S2 = 0,  S3 = 0,  T = 0,  X1 = 0,  X2 = 2
 X3 = 0,  Y1 = 0,  Y2 = 0,  Y3 = 2
 INPUT IS 0  0  2  0  0  2
 AREA IS  2
THE AREA OF THE TRIANGLE IS  2
 *B #2, AT L.160
?
```

Set breakpoint with body at line 240.

Initiate program execution.

Breakpoint detected at line 140; sequence execution initiated.

Program output.

Breakpoint detected at line 160.

Copy breakpoint definitions to TRFILE.

Restore breakpoint definitions contained in TRFILE.

List breakpoint locations.

Initiate program execution.

Figure 5-16. Debug Sessions Illustrating SAVE Command (Sheet 2 of 2)

```
SET HOME P.TRIANGL
SET,BREAKPOINT,L.140 [
DISPLAY,#LINE
LIST,VALUES
]
SET HOME P.TRIANGL
SET BREAKPOINT L.160
SET HOME P.TRIANGL
SET,BREAKPOINT,L.240 [
PRINT "INPUT IS";X1;Y1;X2;Y2;X3;Y3
PRINT "AREA IS ";A
]
```

Figure 5-17. Listing of File TRFILE

The debug sessions in figure 5-18 illustrate the SAVE,GROUP command using the program shown in figure 5-8. The command group MTRX is saved on the file named GRPFILE at the end of the first debug session. At the beginning of the second session, the command READ,GRPFILE restores the group definition. A LINE trap is set at lines 160, 170, and 180 of MATOP. When each trap occurs, the command READ,MTRX is issued to execute the group. Note that this command could have been placed in a body for each breakpoint. The groups would then have been executed automatically.

```
First Debug Session:  Group Definition Saved.

 CYBER INTERACTIVE DEBUG
? set,group,mtrx [ <-------------------- Assign group name and activate collect mode.

 IN COLLECT MODE
? print "printing the contents of m3"
                                          <-------------------- Group body.
? display,#line
? mat print m3

? ]

 END COLLECT
? save,group,grpfile <-------------------- Copy group definition to file GRPFILE.

? quit


Second Debug Session:  Group Definition Restored.

 CYBER INTERACTIVE DEBUG
? read,grpfile <-------------------- Restore group definition contained in GRPFILE.

? list,group,* <-------------------- List current group names and numbers.

 *G #1 = MTRX
? set,trap,line,l.160...l.180 <----------- Set LINE trap.

? go <-------------------- Initiate program execution.

 *T #1, LINE AT L.160
? read,mtrx <-------------------- Initiate execution of the commands in MTRX.

 PRINTING THE CONTENTS OF M3
 #LINE = P.MATOP_L.160
  11            13            15

  17            19            21

  23            25            27


?
```

Figure 5-18. Debug Sessions Illustrating READ and SAVE,GROUP Commands

EDITING A COMMAND SEQUENCE

If you want to make a change to a command sequence in a breakpoint body, trap body, or group, you can remove the definition with the appropriate CLEAR command and reenter the entire sequence. However, this procedure can be time-consuming for lengthy sequences.

CID provides two alternate methods for making changes to a command sequence:

You can save the breakpoint, trap, or group definition on a separate file and edit the file. (You must temporarily exit the current debug session.)

You can turn on veto mode and edit the sequence interactively each time the sequence is executed. See the CYBER Interactive Debug reference manual for an explanation of this method.

SUSPENDING A DEBUG SESSION

CID allows you to suspend the current debug session, return to system command mode, and resume the debug session later. This feature can be used whenever you wish to perform a function outside CID, but it is especially useful for leaving a session to edit a command sequence.

The following commands suspend the current debug session, copy information about the session environment to the specified file, and return control to the operating system:

SUSPEND

This command saves the debug session on the local file ZZZZZDS.

SUSPEND,lfn

This command saves the debug session on the local file specified by lfn.

The information saved on the local file includes a copy of the executing program, all CID internal tables, and all trap, breakpoint, and group definitions. In short, the file contains all the information necessary to continue the debug session. (Program data files are not saved on this local file, but they are unaffected if you do not change their positions or log out.)

The information contained in the file created by a SUSPEND command is intended for CID use only; you should not access it directly. This file preserves the status of a debug session exactly as it existed when the SUSPEND was executed. The suspend file is a local file; however, you can make the file permanent in order to preserve the debug session after a logout. The saved debug session can be resumed in a subsequent terminal session. However, except for sessions involving extremely long programs, you should seldom need to continue a debug session over more than one terminal session.

You should not alter the status of any files used by your program after you issue a SUSPEND command. If you perform any file manipulation operations, such as rewinding files, on files used by your program, you might not be able to restart the session normally.

To resume the suspended debug session, enter one of the following commands:

DEBUG(RESUME)

This command resumes the debug session that was suspended on file ZZZZZDS.

DEBUG(RESUME,lfn)

This command resumes the debug session that was suspended on the file specified by lfn.

The debug session is then restored to its status as it existed at the time of suspension. All breakpoint, trap, and group definitions are restored, and all program and debug variables have the values that existed when SUSPEND was executed.

Remember that the most effective debug sessions are short and simple. Thus, you will seldom need to use the SUSPEND/RESUME capability except to edit command sequences.

EDITING PROCEDURE

To edit a breakpoint body, trap body, or command group, proceed as follows:

1. Save the breakpoint, trap, or group definition with the appropriate SAVE command.
2. Suspend the current session with the SUSPEND command.
3. Use a text editor to make desired changes to the command sequence.
4. Resume the session with the DEBUG(RESUME) command.
5. Remove the old breakpoint, trap, or group definition with the appropriate CLEAR command.
6. Establish the altered definition with the READ command.

After a SUSPEND command, do not modify or change the position of any files used by your program, because the DEBUG(RESUME) command does not restore these files to their status at suspension time.

Examples of the procedures for editing a command sequence under NOS and NOS/BE are shown in figures 5-19 and 5-20, respectively. The purpose of the editing session is to change the command LET Y=2.0 in the group named AGRP to LET Y=3.0. To accomplish this, the group is copied to file SAVFIL, and the debug session is suspended.

Under NOS, the statement XEDIT,SAVFIL calls the system text editor. After the file SAVFIL is printed, the line LET Y=2.0 is found and replaced with LET Y=3.0. When the editing session is terminated with the END command, the debug session is resumed by DEBUG(RESUME). The group is first cleared by the CLEAR,GROUP command and then restored by READ,SAVFIL. See the XEDIT reference manual for detailed information on XEDIT.

Under NOS/BE, the EDITOR control statement calls the system text editor. The command EDIT,SAVFIL,S makes SAVFIL the edit file and assigns a sequence number to each line in the edit file. The command 140=LET Y=3.0 makes the desired change to line 140 of the edit file. The edit file is then copied to file NEWFIL and editing is terminated. See the INTERCOM reference manual for more information on the INTERCOM text editor.

```
 CYBER INTERACTIVE DEBUG
? set,group,agrp [                          ┐
 IN COLLECT MODE                            │
? print "executing group agrp"              │
? display,#line                             ├──── Define group AGRP.
? let x=1.0                                 │
? let y=2.0                                 │
? ]                                         ┘
 END COLLECT
? save,group,savfil,agrp ──────────────────── Copy definition of AGRP to file SAVFIL.
? suspend ─────────────────────────────────── Suspend debug session.

SRU      9.349 UNTS.

 RUN COMPLETE.

xedit,savfil ──────────────────────────────── Call XEDIT program to edit SAVFIL.

 XEDIT 3.1.00
?? print,* ────────────────────────────────── Print edit file.
SET,GROUP,AGRP [
PRINT "EXECUTING GROUP AGRP"
DISPLAY,#LINE
LET X=1.0
LET Y=2.0
]
END OF FILE
?? next 4 ─────────────────────────────────── Advance line pointer.
LET Y=2.0
?? replace ────────────────────────────────── Replace current line.
? let y=3.0
?? print
LET Y=3.0
?? end ────────────────────────────────────── Leave edit mode.
SAVFIL  IS A LOCAL FILE

 READY.

debug(resume) ─────────────────────────────── Resume debug session.

 CYBER INTERACTIVE DEBUG RESUMED
? clear,group,agrp ────────────────────────── Remove old definition of AGRP.
? read,savfil ─────────────────────────────── Establish new definition of AGRP.
? list,group,agrp ─────────────────────────── List definition of AGRP.
 *G #1 = AGRP
 SET,GROUP,AGRP [
 PRINT "EXECUTING GROUP AGRP"
 DISPLAY,#LINE
 LET X=1.0
 LET Y=3.0
 ]
?
```

Figure 5-19. Editing a Command Sequence Under NOS

```
CYBER INTERACTIVE DEBUG
?set,group,agrp [                   ┐
                                    │
IN COLLECT MODE                     │
?print "executing group agrp"       │
                                    │
?display,#line                      ├──── Define group AGRP.
                                    │
?let x=1.0                          │
                                    │
?let y=2.0                          │
                                    │
?]                                  ┘

END COLLECT
```

Figure 5-20. Editing a Command Sequence Under NOS/BE (Sheet 1 of 2)

```
?save,group,savfil,agrp <------------------ Copy definition of AGRP to file SAVFIL.

?suspend <---------------------------------- Suspend debug session.

  DEBUG SUSPENDED
COMMAND- editor <--------------------------- Call editor.

..edit,savfil,s <--------------------------- Make SAVFIL the edit file.

..list,all <-------------------------------- List edit file.


   100=SET,GROUP,AGRP [
   110=PRINT "EXECUTING GROUP AGRP"
   120=DISPLAY,#LINE
   130=LET X=1.0
   140=LET Y=2.0
   150=]
..140=let y=3.0 <--------------------------- Replace line 140.
save,newfil,noseq <------------------------- Copy edit file to NEWFIL.

..bye <------------------------------------- Leave EDITOR.

COMMAND- debug(resume) <-------------------- Resume debug session.

CYBER INTERACTIVE DEBUG RESUMED
?clear,group,agrp <------------------------- Remove old definition of AGRP.

?read,newfil <------------------------------ Establish new definition of AGRP by reading NEWFIL.

?list,group,agrp <-------------------------- List definition of AGRP.

*G #1 = AGRP
SET,GROUP,AGRP [
PRINT "EXECUTING GROUP AGRP"
DISPLAY,#LINE
LET X=1.0
LET Y=3.0
]
?
```

Figure 5-20. Editing a Command Sequence Under NOS/BE (Sheet 2 of 2)

INTERRUPTING AN EXECUTING SEQUENCE

A terminal interrupt allows you to gain control at any time during a debug session. If your program is executing at the time of the interrupt, the INTERRUPT trap occurs as described in section 3. However, if a command sequence is executing at the time of the interrupt, execution of the sequence is suspended and CID displays the message:

```
INTERRUPTED
?
```

You respond as follows:

OK or YES

CID resumes sequence execution at the point of the interrupt.

GO or NO,SEQ

CID disregards all remaining commands in the sequence and resumes execution of the program.

Any CID command

CID executes the specified command and resumes execution of the sequence at the point of interrupt.

If CID is in the process of displaying information when the interrupt occurs, the information remaining to be printed is lost. A terminal interrupt is therefore an effective means of stopping excessive CID output.

COMMAND SEQUENCE EXAMPLE

Following is an example of a debug session that uses command sequences. The program CORRBSC, debugged in section 3, is used to illustrate how sequences can be used to speed up the debugging process.

The original version of CORRBSC, with errors, is shown in figure 3-23 (see section 3). Several debug sessions were required to debug the program completely. Commands issued during one session had to be reentered in subsequent sessions. This example demonstrates how this repetition can be eliminated by including the assignment commands in breakpoint and trap bodies and saving the breakpoint and trap definitions on a separate file for use in later sessions. The example also demonstrates how an appropriate command sequence can be used to simulate the reading of input data.

The NOS text editor is used to create the three command files shown in figure 5-21. Each file corresponds to a test case. The files contain assignment commands that insert the correct value for M and test values in the arrays X and Y. Two commands separated by a semicolon are included on each line except the first. The files are named BTST1, BTST2, and BTST3, respectively.

Another file named BFILE, shown in figure 5-22, is created using the text editor. This file contains three breakpoint definitions. The first breakpoint is set at line 220. The PAUSE command will temporarily suspend execution of the breakpoint body. A READ command can then be issued to execute the commands in BTST1. The command GOTO 270 will resume program execution at line 270, skipping the READ statement. The second breakpoint is set at line 380. The body of this breakpoint contains assignment commands to calculate the correct values for S5 when the body is executed. The third breakpoint is set at line 400. The PAUSE command will suspend execution of the breakpoint body if D is equal to zero.

The debug session for program CORRBSC is shown in figure 5-23. At the beginning of the session, the command READ,BFILE is issued to establish the breakpoint definitions, the command LIST,BREAKPOINT,#1,#2,#3 is entered to list the breakpoints and their bodies, and the command GO is entered to initiate program execution. When execution of the PAUSE command suspends program execution, a READ command is issued to load the arrays X and Y. Execution is resumed by the GO command, and the commands in the sequences are executed automatically. Note that only one debug session is needed to run all three test cases. When the END trap occurs after the first and second runs, the GOTO command is used to start program execution at line 150 for the next test case.

```
Listing of BTST1:

     LET M=5
     LET X(1)=1.0;LET Y(1)=1.0
     LET X(2)=10.0;LET Y(2)=10.0
     LET X(3)=7.6;LET Y(3)=7.6
     LET X(4)=2.9;LET Y(4)=2.9
     LET X(5)=5.1;LET Y(5)=5.1


Listing of BTST2:

     LET M=5
     LET X(1)=3.0;LET Y(1)=1.0
     LET X(2)=3.0;LET Y(2)=5.1
     LET X(3)=3.0;LET Y(3)=7.6
     LET X(4)=3.0;LET Y(4)=10.0
     LET X(5)=3.0;LET Y(5)=15.0


Listing of BTST3:

     LET M=5
     LET X(1)=0.0;LET Y(1)=0.0
     LET X(2)=0.0;LET Y(2)=100.0
     LET X(3)=0.0;LET Y(3)=0.0
     LET X(4)=0.0;LET Y(4)=500.0
     LET X(5)=0.1;LET Y(5)=10.0
```

Figure 5-21. Command Files for Program CORRBSC

```
SET,BREAKPOINT,L.220 [
PAUSE,"INPUT?"
DISPLAY,#LINE
PRINT "PRINTING THE CONTENTS OF ARRAYS X AND Y"
MAT PRINT X,Y
GOTO 270
]
SET,BREAKPOINT,L.380 [
PRINT "EXECUTING BREAKPOINT AT L.380"
LET S5=X(1)*Y(1)+X(2)*Y(2)+X(3)*Y(3)
LET S5=S5+X(4)*Y(4)+X(5)*Y(5)
]
SET,BREAKPOINT,L.400 [
PRINT "EXECUTING BREAKPOINT AT L.400"
PRINT "N=";N,"D=";D
IF D=0.0 THEN
PAUSE,"D IS 0"
]
```

Figure 5-22. Listing of File BFILE

```
 CYBER INTERACTIVE DEBUG
? read,bfile                                    Establish breakpoint definitions
                                                stored in BFILE.
? list,breakpoint,*                             List breakpoint locations.

 *B #1 = L.220 ,   *B #2 = L.380 ,   *B #3 = L.400
? list,breakpoint,#1,#2,#3                      List breakpoint bodies.

 *B #1 = L.220
 SET,BREAKPOINT,L.220 [
 PAUSE,"INPUT?"
 DISPLAY,#LINE
 PRINT "PRINTING THE CONTENTS OF ARRAYS X AND Y"   Breakpoint #1.
 MAT PRINT X,Y
 GOTO 270
 ]
 *B #2 = L.380
 SET,BREAKPOINT,L.380 [
 PRINT "EXECUTING BREAKPOINT AT L.380"
 LET S5=X(1)*Y(1)+X(2)*Y(2)+X(3)*Y(3)            Breakpoint #2.
 LET S5=S5+X(4)*Y(4)+X(5)*Y(5)
 ]
 *B #3 = L.400
 SET,BREAKPOINT,L.400 [
 PRINT "EXECUTING BREAKPOINT AT L.400"
 PRINT "N=";N,"D=";D
 IF D=0.0 THEN                                   Breakpoint #3.
 PAUSE,"D IS 0"
 ]
? go
                                                Breakpoint detected at line 220;
 *B #1, AT L.220                                sequence execution initiated.
 INPUT?                                         PAUSE command suspends execution.
? read,btst1                                    Initiate execution of commands in
                                                BTST1.
? go                                            Resume sequence execution.

 #LINE = P.CORRBSC L.220
 PRINTING THE CONTENTS OF ARRAYS X AND Y
  1             10            7.6           2.9           5.1


  1             10            7.6           2.9           5.1


 EXECUTING BREAKPOINT AT L.380                  Breakpoint #2 is executed.
 EXECUTING BREAKPOINT AT L.400                  Breakpoint #3 is executed.
 N= 66739.6      D= 66739.6
CORRELATION COFFICIENT = 1                      Program prints result.
 *T #17, END IN L.420                           Execution terminates.
? goto 150                                      Resume execution at line 150.

 *B #1, AT L.220                                Breakpoint detected at line 220.
 INPUT?
? read,btst2                                    Initiate execution of commands in
                                                BTST2.
```

Figure 5-23. Debug Session for Program CORRBSC Using Command Sequences (Sheet 1 of 2)

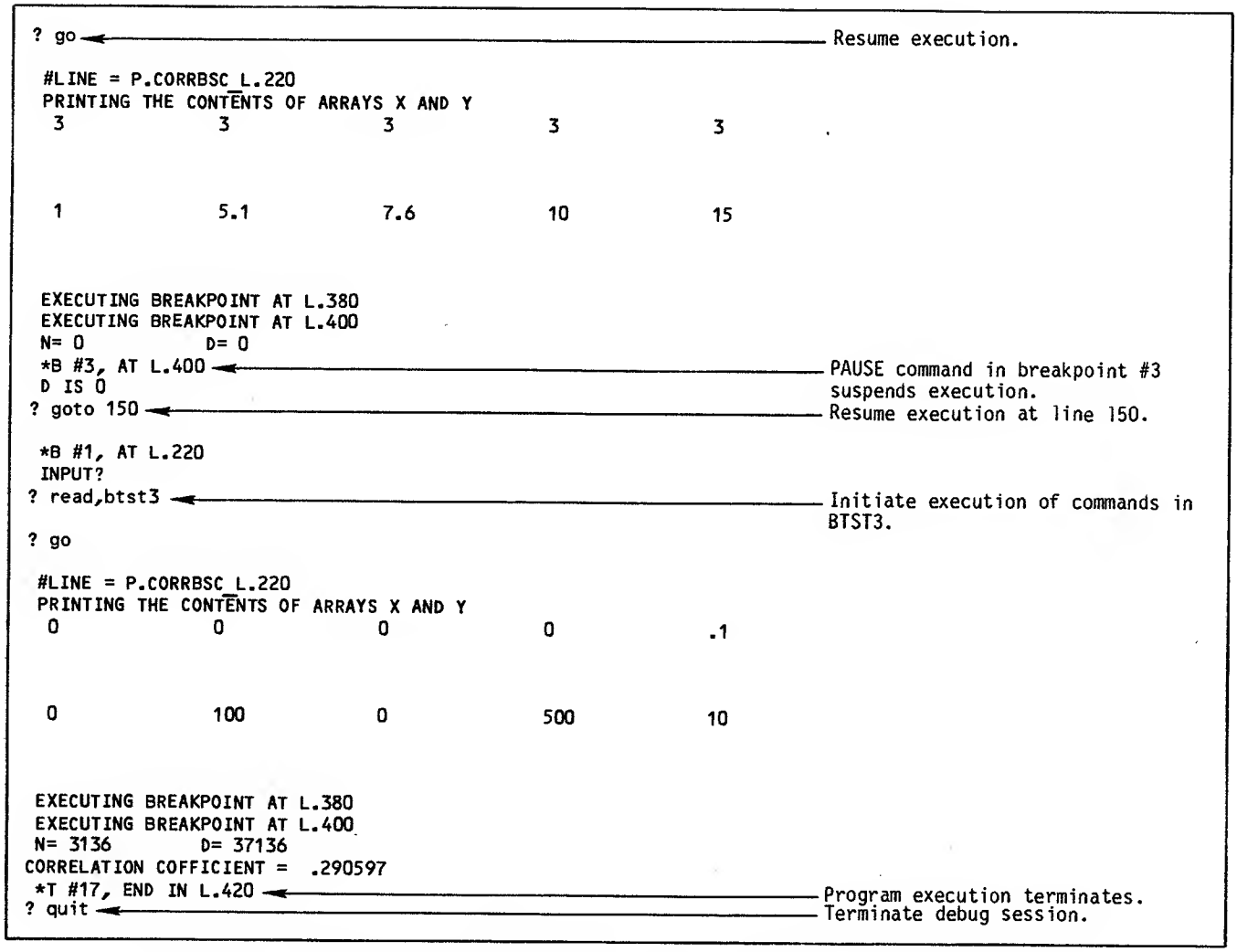

Figure 5-23. Debug Session for Program CORRBSC Using Command Sequences (Sheet 2 of 2)

STANDARD CHARACTER SETS

A

Control Data operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect throughout the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1; 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of table A-1 are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

TABLE A-1. STANDARD CHARACTER SETS

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(	0-8-4	34	(	12-8-5	050
52	)	12-8-4	74	)	11-8-5	051
53	$	11-8-3	53	$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[	8-7	17	[	12-8-2	133
62	]	0-8-2	32	]	11-8-2	135
63	% ††	8-6	16	% ††	0-8-4	045
64	≠	8-4	14	" (quote)	8-7	042
65	→	0-8-5	35	_ (underline)	0-8-5	137
66	∨	11-0	52	!	12-8-7	041
67	∧	0-8-7	37	&	12	046
70	↑	11-8-5	55	' (apostrophe)	8-5	047
71	↓	11-8-6	56	?	0-8-7	077
72	<	12-0	72	<	12-8-4	074
73	>	11-8-7	57	>	0-8-6	076
74	≤	8-5	15	@	8-4	100
75	≥	12-8-5	75	\	0-8-2	134
76	¬	12-8-6	76	^ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

GLOSSARY

B

Abort -
To terminate a program or job when an error condition (hardware or software) exists from which the program or computer cannot recover.

Auxiliary File -
A file, established by the SET,AUXILIARY command, to which CYBER Interactive Debug (CID) output is written. The output types written to this file are specified by special output codes.

Batch Mode -
A mode of CID execution which allows programs intended for batch execution to be executed under CID control.

Breakpoint -
A designated location in a program where execution is to be suspended.

Collect Mode -
A mode of CID execution in which commands you enter are not executed, but are included in a group, trap, or breakpoint body. Collect mode is initiated by a left bracket ([) at the end of a SET,TRAP, SET,GROUP, or SET,BREAKPOINT command, and is terminated by a right bracket (]).

Debug Mode -
A mode of execution in which special CID tables are generated during compilation and in which your programs are executed under CID control; initiated by a DEBUG(ON) control statement, and terminated by a DEBUG(OFF) control statement.

Debug Session -
A sequence of interactions between you and CID, beginning when execution of your program is initiated in debug mode and ending when a QUIT command is issued.

Group -
A sequence of CID commands established and assigned a name by a SET,GROUP command and executed when a READ command is issued.

Home Program -
Program unit in which variables and line numbers you reference in CID commands are assumed to be located unless appropriate qualifiers appear. By default, the home program is the program unit being executed at the time CID gains control. You can change the default with the SET,HOME command.

Interactive -
Job processing in which you and the system communicate with each other, rather than processing in which you submit a job and receive output later.

Interactive Mode -
The normal mode of CID execution. You enter commands directly from the terminal and CID immediately executes the commands. CID can also execute in batch mode.

Interpret Mode -
A mode of execution in which a special routine, called an interpreter, examines each machine instruction to be executed in your program and simulates its execution by the execution of several of its own instructions. Execution in interpret mode consequently takes 20 to 50 times as long as direct execution. Certain CID features require interpret mode execution.

Interrupt (noun) -
A control signal that you issue from the terminal. If your program is executing when CID detects an interrupt, an INTERRUPT trap occurs; if a CID command sequence is executing, the command sequence is suspended and you gain control.

Under NOS, CID interprets both the user-break-1 and the user-break-2 terminal keys as the interrupt key. The user-break-1 and user-break-2 keys differ depending on the terminal type (see the NOS Version 2 reference set, Volume 3). On most terminals these keys are CONTROL P and CONTROL T, respectively. You can issue an interrupt by pressing CONTOL P (or CONTROL T) followed by a carriage return.

Under NOS/BE, you can issue an interrupt by pressing %A followed by a carriage return (see the INTERCOM reference manual).

Interrupt (verb) -
To stop a running program in such a way that it can be resumed at a later time.

Program Unit -
A BASIC main program or FORTRAN subroutine.

Terminal Session -
The sequence of interactions between you and a terminal which begins when you log in and terminates when you log out. Contrast with Debug Session.

Trap (noun) -
A mechanism that detects the occurrence of a specified condition, suspends execution of your program at that point, and transfers control to CID.

Trap (verb) -
To suspend program execution and transfer control to CID upon the detection of a specified condition.

*BATCH MODE DEBUGGING

C

CYBER Interactive Debug (CID) is primarily intended to be used interactively, but can be used in batch mode. Possible reasons for using batch mode include the possibility of a large volume of output, or lack of access to a terminal. In batch mode, however, you must plan the entire session in advance. This requires care and a knowledge of what errors are likely to occur.

To conduct a debug session in batch mode, commands must exist on a file of card images called DBUGIN from which CID reads all input. You can create this file by using the system text editor, or you can punch the commands on cards, include them as part of the job deck, and copy file INPUT to DBUGIN. Commands are punched or written in the same format as in interactive mode; a card can contain a single command or multiple commands separated by semicolons.

As in interactive execution, debug mode is established by the DEBUG control statement. The debug session is initiated by a statement to load and execute the program. Control transfers immediately to CID, which begins executing the commands in DBUGIN. When CID encounters a GO or EXECUTE in the command stream, control transfers to your program. Your program executes until a breakpoint or trap is encountered. In this manner, control transfers between your program and CID with no intervention from you.

A QUIT command is normally the last command of the sequence. However, this command can be omitted and CID will terminate after the last command has been executed.

Following are some restrictions that apply to batch mode debugging:

- Invalid commands are disregarded; when CID encounters such a command, processing continues with the next command.
- Commands that would generate a warning message in interactive mode are executed in batch mode.
- All commands are executed except when execution is impossible; you cannot establish veto mode in a batch session.

In batch mode, all output from CID is written to a file named DBUGOUT. This is a local file and it is the user's responsibility to print the file or make it permanent. You can control the types of output sent to DBUGOUT with the SET,OUTPUT command. Output can also be sent to a separate file with the SET,AUXILIARY command.

Batch output from a debug session does not normally show the CID commands you specified as they are executed. CID reads the commands from DBUGIN but does not echo them to DBUGOUT unless the T option is specified on the SET,OUTPUT command. Use of this option usually improves the readability of a batch debug session.

With the exception of the SET,VETO command, all CID commands are valid in batch mode. You can set breakpoints and traps, define command sequences, display and alter the values of program variables, and resume program execution. The commands in DBUGIN should be specified in the same order as in interactive mode. CID accesses DBUGIN for all input that would normally be input from the terminal.

A suggested technique for batch mode debugging is to use only breakpoints and traps with bodies. This way, the commands to be executed on suspension of execution appear in the input stream immediately after the SET,BREAKPOINT or SET,TRAP command that caused suspension. In addition, only one GO command is required.

An example of a program to be debugged in batch mode (under NOS) is illustrated in figure C-1. (To execute this program under NOS/BE, replace the job, user, and charge statements with a job statement containing the appropriate accounting information.) Breakpoints with bodies are set initially at lines 110 and 130, and program execution is initiated. When the first breakpoint is encountered, CID receives control, executes commands in the body, and returns control to the program. The command GOTO 120 skips the BASIC statement that reads in the data. When the breakpoint at line 130 is encountered, CID executes the LIST,VALUES and QUIT commands. The contents of the output file DBUGOUT are shown in figure C-2.

```
JOB statement
USER statement
CHARGE statement
COPYBR(INPUT,DBUGIN)
REWIND(DBUGIN)
DEBUG(ON)
BASIC.
REWIND(DBUGOUT)
COPYBF(DBUGOUT,OUTPUT)
7/8/9 in column 1
SET,OUTPUT E,W,D,I,T
SET,BREAKPOINT,L.110 [
LET X1=0.0;LET Y1=0.0
LET X2=2.0;LET Y2=0.0
LET X3=1.0;LET Y3=-1.0
GOTO 120
]
SET,BREAKPOINT,L.130 [
LIST,VALUES
QUIT
]
GO
7/8/9 in column 1
00100 REM PROGRAM NTRIAN
00110 READ X1,Y1,X2,Y2,X3,Y3
00120 IF X1=-999 THEN STOP
00130 GOSUB 00170
00140 PRINT "THE AREA OF THE TRIANGLE IS ";A
00150 GOTO 00110
00160 REM SUBROUTINE AREA
00170 LET S1=SQR((X2-X1)^2+(Y2-Y1)^2)
00180 LET S2=SQR((X3-X1)^2+(Y3-Y1)^2)
00190 LET S3=SQR((X3-X2)^2+(Y3-Y2)^2)
00200 LET T=(S1+S2+S3)/2.0
00210 LET A=SQR(T*(T-S1)*(T-S2)*(T-S3))
00220 RETURN
00230 DATA 0.0,0.0,2.0,0.0,0.0,2.0
00240 DATA -999,0,0,0,0,0,0
00250 END
6/7/8/9 in column 1
```

Figure C-1. Card Deck for Batch Debug Session

```
          CYBER INTERACTIVE DEBUG 1.2-552.        81/12/18. 09.54.41.     PAGE   1

CYBER INTERACTIVE DEBUG
SET,OUTPUT E,W,D,I,T
SET,BREAKPOINT,L.110 [
IN COLLECT MODE
LET X1=0.0;LET Y1=0.0
LET X2=2.0;LET Y2=0.0
LET X3=1.0;LET Y3=-1.0
GOTO 120
]
END COLLECT
SET,BREAKPOINT,L.130 [
IN COLLECT MODE
LIST,VALUES
QUIT
]
END COLLECT
GO
P.BASICXX
A =  0,   S1 =  0,   S2 =  0,   S3 =  0,   T =  0,   X1 =  0,   X2 =  2,   X3 =  1,
Y1 =  0,   Y2 =  0,   Y3 = -1
```

Figure C-2. Listing of File DBUGOUT

SUMMARY OF CID COMMANDS

D

Table D-1 summarizes the CYBER Interactive Debug (CID) commands described in this guide, and specifies the page where more detailed information can be obtained.

TABLE D-1. CID COMMAND SUMMARY

Command	Short Form	Described on Page	Description
CLEAR,AUXILIARY	CAUX	4-3	Closes the auxiliary output file.
CLEAR,BREAKPOINT	CB	3-3	Removes breakpoints.
CLEAR,GROUP	CG	5-5	Removes command group definitions.
CLEAR,OUTPUT	COUT	4-3	Turns off output to the terminal.
CLEAR,TRAP	CT	3-10	Removes traps.
DEBUG(RESUME)		5-18	Resumes the suspended debug session.
DISPLAY	D	3-15	Displays the contents of program locations.
EXECUTE	EXEC	5-10	Resumes execution of your program.
GO		2-3 5-10	Resumes execution of your program or of a suspended command sequence.
GOTO		3-17	Resumes execution of your program at a specified line number.
IF		5-12	Provides for conditional execution of CID commands.
JUMP		5-12	Causes a transfer of control within a command sequence.
LABEL		5-12	Designates a label to be used as the destination of a JUMP command.
LET		3-16	Replaces the value of the expression on the right of the equal sign with the current value of the variable on the left of the equal sign.
LIST,BREAKPOINT	LB	3-3	Displays information about breakpoints defined for a debug session.
LIST,GROUP	LG	5-5	Displays information about command groups defined for a debug session.
LIST,MAP	LM	4-7	Displays load map information.
LIST,STATUS	LS	3-19	Displays information about the current status of the debug session.
LIST,TRAP	LT	3-9	Displays information about traps defined for a debug session.
LIST,VALUES	LV	3-14	Displays names and values of program variables.
MAT PRINT		3-13	Displays the contents of program arrays.
PAUSE		5-10	Suspends execution of a command sequence.
PRINT		3-13	Displays the contents of program variables.

TABLE D-1. CID COMMAND SUMMARY (Contd)

Command	Short Form	Described on Page	Description
QUIT		2-3	Terminates the debug session.
READ		5-5 5-14	Executes a group or file sequence or trap, breakpoint, and group definitions saved on a file.
SAVE,BREAKPOINT	SAVEB	5-14	Writes breakpoint definitions to a file.
SAVE,GROUP	SAVEG	5-14	Writes group definitions to a file.
SAVE,TRAP	SAVET	5-14	Writes trap definitions to a file.
SET,AUXILIARY	SAUX	4-3	Establishes an auxiliary output file.
SET,BREAKPOINT	SB	3-2	Defines a breakpoint.
SET,GROUP	SG	5-4	Defines a group.
SET,HOME	SH	4-7	Designates a home program.
SET,INTERPRET,OFF	SI OFF	3-11	Turns off interpret mode.
SET,INTERPRET,ON	SI ON	3-11	Turns on interpret mode.
SET,OUTPUT	SOUT	4-2	Selects output types to be displayed at the terminal.
SET,TRAP,LINE	ST L	3-7	Defines a LINE trap.
SET,TRAP,STORE	ST S	3-8	Defines a STORE trap.
STEP	S	4-1	Executes a few lines at a time.
SUSPEND		5-17	Suspends the debug session.

INDEX

COMMENT SHEET

MANUAL TITLE: CYBER Interactive Debug Version 1 Guide for Users of BASIC Version 3

PUBLICATION NO.: 60484110 REVISION: A

NAME:

COMPANY:

STREET ADDRESS:

CITY: STATE: ZIP CODE:

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

___ Please reply ___ No reply necessary

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

215 Moffett Park Drive
Sunnyvale, California 94086

FOLD

FOLD

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S

CONTROL DATA CORPORATION